

NEW ABSTRACTIONS FOR MOBILE
CONNECTIVITY AND RESOURCE
MANAGEMENT

ROBERT RANDOLPH KIEFER

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE
ADVISER: PROFESSOR MICHAEL FREEDMAN

MAY 2016

© Copyright by Robert Randolph Kiefer, 2016.

All rights reserved.

Abstract

Mobile devices have become an important—and for many, primary—means of connecting to the Internet. They offer great flexibility in how and where they are used, even with constraints like battery life and data caps. However, their network stack and operating system are lacking when it comes to network mobility and managing their limited resources. First, mobility, if even addressed by devices and apps, is done in an ad-hoc manner to “hide” connectivity issues rather than fix underlying shortcomings. Second, resource management is dealt with in broad strokes, scattered among various app and device settings pages.

In this thesis, we provide new abstractions addressing these issues. To better handle mobility in the network stack, we need to remove the overloading of IP/port as both app-level connection identifiers and network addresses, thereby allowing apps to maintain a connection even when addresses change. To that end, we present ECCP, a protocol that splits the current transport layer into two: one for connection management (e.g., startup, teardown, migration) and one for data delivery semantics (e.g., reliable, best-effort, etc). ECCP is part of a larger network architecture, Serval, which addresses several pain points with today’s networked systems, consisting of replicated backend services and mobile, multi-homed clients. We derive a state machine for ECCP supporting migration and multipath, and through Serval, we demonstrate ECCP’s value in scenarios utilizing mobility (e.g., VM migration) while achieving par performance with TCP.

To utilize better mobility support, devices need a more expressive resource management system. Currently, network choice and resource management is limited to broad decisions (e.g., if cellular network usage is allowed). Users must either choose “good enough” settings or else micromanage their device and apps, e.g., turning WiFi on/off while streaming music between hotspots. Instead, we introduce Tango, a programmatic, policy-based network resource management platform, that works at the

device and app levels to offer more flexibility. Users and apps define policies that dynamically monitor device state to adjust resource usage on their behalf. Using Tango, a user can, e.g., minimize cellular data while streaming music (using it only to prevent playback pauses), saving up to 60-95%.

Bibliographic Notes

The research presented in this thesis is based on work performed in collaboration with my adviser Michael Freedman, as well as Erik Nordström, David Shue, Prem Gopalan, Matvey Arye, Steven Y. Ko, and Jennifer Rexford. The protocol for incorporating mobility and multipath into the network stack introduced in Chapter 3 appears in a paper co-authored with Matvey Arye, Erik Nordström, Jennifer Rexford, and Michael Freedman [2]. The implementation of that protocol as part of a larger network architecture that is introduced in Chapter 3 and evaluated in Chapter 4 appears in a paper co-authored with Erik Nordström, David Shue, Prem Gopalan, Matvey Arye, Steven Y. Ko, Jennifer Rexford, and Michael Freedman [24]. The policy platform for resource management introduced in Chapter 5 appears in a paper co-authored with Erik Nordström and Michael Freedman [19].

Acknowledgements

I would like to thank my adviser, Professor Michael Freedman, for his invaluable guidance. He was always able to focus me on my best ideas and on the core contributions and benefits of my work. His insight into a variety of problem spaces and never-ending ability to ask just the right question helped me refine my work to what it is today. I have learned a great deal on approaching complex problem spaces and zeroing in on the important parts, and for that I have him to thank for that. However little I knew before has been greatly expanded under his guidance.

My committee members—Jen Rexford, Nick Feamster, David August, and Margaret Martonosi—have also been valuable in molding this thesis into its final form by giving me new questions to consider and ideas to flesh out. I appreciate the time they took to review my work and give me their thoughtful considerations at its various stages.

Many thanks also to my undergraduate adviser, Professor Bobby Bhattacharjee, and members of his lab, especially Dave Levin. They took me in when I was not sure what I wanted to do with my career and got me interested in doing research. Many late nights working on projects helped give me a taste of what graduate school was about.

Huge thanks to my colleagues, collaborators, and fellow graduate students at Princeton. Whether it was debating how to best present our work, seeing who could best throw magnets at a white board, or just enjoying lunch, a beer, or some simple down time, it was a great pleasure and filled with memories. In particular, I want to recognize Erik Nordstrom, a collaborator on all my publications, who spent plenty of anxious hours before a deadline helping make our ideas come across more elegantly. Other special thanks to Dave Shue, Logan Stafman, Wyatt Lloyd, Matvey Arye, Jude Nelson, and many others for all the laughs and good times that helped take us away from our work.

Finally, thank you to my friends and family. All the road trips back and forth from Maryland were worth it to see you. My parents, Randy and Lynn, thank you for believing in me and encouraging me to always be my best. My siblings, Beth and Matt, thank you for enjoyable trips home and for sending me photos of my niece and my dogs, which never failed to make me smile. Thank you to all my friends, who even though I lived somewhat far away, still invited me to everything like I was still in town. And a special thank you to my girlfriend, Tessy, who has been incredibly encouraging and enthusiastic as I neared the finish line. Her unflinching willingness to help made all of this much easier.

The research in this thesis was funded by Office of Naval Research N00014-09-1-0910: Towards a Service-Centric Network Architecture for Fault Tolerance, Migration, and Mobility; National Science Foundation CNS-0904729: ARRA: NeTS Medium: A SCAFFOLD for Service-Centric Networking; National Science Foundation CNS-1040708: FIA: Collaborative Research: NEBULA: A Future Internet That supports Trustworthy Cloud Computing; and National Science Foundation IIS-1250990: BIGDATA: Small: DCM: JetStream: A Flexible Distributed System for Online and In-Place Data Analysis.

To my friends and family.

Contents

Abstract	iii
Bibliographic Notes	v
Acknowledgements	vi
List of Tables	xii
List of Figures	xiv
List of Pseudocodes	xvii
1 Introduction	1
1.1 Mobility and the Network	2
1.2 Device Resource Management	4
2 Background	8
2.1 Supporting Network Mobility	8
2.2 Handling Limited Resources	11
3 Supporting Mobility	15
3.1 Protocol Requirements and Related Work	16
3.1.1 Protocol Requirements	16
3.1.2 Related Work	17
3.2 The ECCP Protocol	20
3.2.1 Establishing a New Connection With a Single Flow	21
3.2.2 Adding Flows to an Existing Connection	23

3.2.3	Changing the IP Addresses of Existing Flows	25
3.3	Other Concerns	28
3.3.1	Verification	28
3.3.2	Security	29
3.3.3	Simultaneous Movement	30
3.4	Bigger Picture: Serval	31
3.4.1	Naming Abstractions	32
3.4.2	The Serval Network Stack	35
4	ECCP/Serval Evaluation	42
4.1	Serval Prototype	42
4.1.1	Lessons From the Serval Prototype	42
4.1.2	Performance Microbenchmarks	44
4.1.3	Application Portability	45
4.2	Experimental Case Studies	46
4.3	Incremental Deployment	49
5	Managing Limited Resources with Tango	52
5.1	Motivation and Challenges	54
5.1.1	Balancing Costs, Caps, and Battery	55
5.1.2	Ensuring Good User Experience	56
5.1.3	Managing Conflicting Interests	57
5.2	Tango Design	58
5.2.1	The Controller and Policy Execution	60
5.2.2	A Programmatic Approach to Policy	63
5.2.3	Discussion: Policy in Practice	67
5.3	Tango Related Work	69

6	Tango Evaluation	72
6.1	Tango Prototype	72
6.2	Case Study Evaluation	74
6.2.1	Experimental Setup	74
6.2.2	Case Study 1: Music Streaming	79
6.2.3	Case Study 2: Policy Across Apps	85
7	Conclusion	90
7.1	Future Work	91
	Bibliography	93

List of Tables

3.1	Comparison of ECCP with alternative approaches	17
3.2	State stored by ECCP for connections and flows	22
3.3	Comparison of BSD socket protocol families: INET sockets (e.g., TCP/IP) use both IP address and port number, while Serval simply uses a serviceID.	37
4.1	TCP throughput of the native TCP/IP stack, the Serval stack, and the two stacks connected through a translator. UDP routing throughput of native IP forwarding and the Serval stack.	44
4.2	Applications currently ported to Serval. Codebase size and Serval changes measured in lines of code.	46
5.1	Application settings available for managing resource usage.	53
5.2	Device state sources and metrics.	60
5.3	Actions on interfaces and flows. App policy can only perform flow actions and only on their flows.	66
6.1	WiFi connectivity quality statistics across many traces of the same path. “Good” signifies both upstream and downstream had drop rates $\leq 10\%$	77
6.2	Evaluated policies.	79

6.3	<code>Unl</code> and <code>Rate</code> keep cellular active and drain battery faster, while <code>App</code> is able to reduce the drain.	85
-----	--	----

List of Figures

1.1	The TCP/IP transport layer broken up into two sublayers, with ECCP operating below data delivery.	3
3.1	The ECCP state machine. Double lines indicate transitions that create a new subflow and associated control block (state). Asterisks indicate that the receiving state must be able to handle getting duplicate messages in an idempotent manner.	20
3.2	A conceptual view of a packet, showing the location and composition of ECCP headers. The data delivery header is typically a regular transport header (e.g., TCP), although used only to provide data delivery functionality.	21
3.3	The ECCP protocol for establishing a new connection, using a three-way handshake similar to TCP. However, ECCP endpoints also share which interfaces they allow flows on and use flowIDs instead of IP/port.	23
3.4	Adding a new flow to an existing connection in ECCP.	24
3.5	Choosing a different interface for a new flow in ECCP.	25
3.6	The ECCP protocol for changing the address associated with an already established flow. FlowId _m and FlowId _s are the IDs for the flow, while A5 is the new address and (A5,A6) is the new interface list.	27
3.7	An example of a misbehaving protocol trace when using implicit ACKs to confirm migration.	28

3.8	New Serval identifiers visible in packets, between the network and transport headers. Some additional header fields (e.g., checksum, length, etc.) are omitted for readability.	33
3.9	Serval network stack with service-level control/data plane split. . . .	36
3.10	Serval forwarding between two end-points through an intermediate service router (SR).	38
3.11	Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.	41
4.1	A Serval-enabled host migrates one of the flows sharing a GigE interface to a second interface, yielding higher throughput for both.	47
4.2	A VM migrates across subnets, causing a short interruption in the data flow.	48
4.3	The thesis author uses a Serval-enabled phone to stream music while walking across campus. The phone migrates the connection between available WiFi (red) and cellular 4G (gray) networks, without loss of playback quality. The opacity of each data point is an indicator of the throughput (normalized within network type) achieved at that location.	49
5.1	The Tango architecture. Device state is continuously collected from sources in the device kernel, packaged by the controller (center gray box), and evaluated by all registered policies. Each policy outputs a list of actions (e.g., rate limit) called plans, which are carried out by the controller using tools provided by the kernel and operating system.	58
6.1	Emulations of a campus walk. Both CBR traffic (a) and TCP traffic (b) emulations reproduce the real world patterns with high fidelity. .	76

6.2	Interactions between TCP timers and changing connectivity. In (a), a lost TCP ACK packet (just before 350s) causes a 80s TCP timeout, leading to a missed opportunity to use good connectivity. In (b) some packets get through prior to 350s, causing a shorter timeout that allows use of the connectivity.	77
6.3	Distribution of durations of low TCP throughput, Android versus Tango switching. CBR UDP traffic serves as a baseline showing connectivity.	78
6.4	Effect of aggressive WiFi offloading on playback buffers. Android's tendency to persist on WiFi, despite no TCP progress, leaves little room for policy to play a role in improving the application experience.	80
6.5	Buffer usage of different Tango policies. All avoid any pauses during playback.	82
6.6	Network usage showing how Tango policies (Rate and App) can drastically reduce cellular usage compared to unlimited usage (Unl). . . .	83
6.7	Unl and Rate drain battery faster due to keeping the cell link active, while App is able to reduce the drain.	84
6.8	App-level fairness at link level, while app policy optimizing performance given constraints.	86
6.9	Providing priority dynamically to foreground traffic for better user experience.	87
6.10	Insufficient resource isolation in today's phones: background music reduces web performance.	88
6.11	Providing dynamic priority between music and web. The music app hints at its need for the network to provide improved page load times when its buffer is healthy (dark gray) and minimal disruption when its buffer needs replenishing (light gray).	89

List of Pseudocodes

1	Tango Control Loop	59
2	App policy with hints	64
3	User policy with app hints	64
4	Avoid poor WiFi	66
5	Prioritize foreground app	68

Chapter 1

Introduction

Today’s most common computing devices increasingly are mobile devices. These devices — smartphones, tablets, and laptops — offer nearly ubiquitous connectivity for users on the move. Yet, the experience can be lackluster because the network stack and the operating systems they run are not fine-tuned for mobile use cases. Because of this, these devices’ greatest strengths (e.g., mobility and being multi-homed) are diminished, and their greatest weaknesses (i.e., limited resources such as battery life and data caps) are exacerbated.

The typical TCP/IP network stack used by mobile devices was designed without mobility and multiplicity in mind. When it was designed, devices were stationary and usually single-homed, and without the expectation a device would change its network mid-communication. But for mobile devices, changing networks is not a rarity; users often pass through multiple WiFi access points and/or cell networks when on the move. Unfortunately, the current network stack overloads network identifiers (e.g., IP addresses and ports) as not only a network identifier, but also as part of the “five-tuple” that hosts use to demux and identify a connection. Despite having multiple networks to utilize, these devices’ utilization of these networks is subpar. Typically devices prefer WiFi to other networks; eagerly switching off cellular even if the WiFi

network signal is poor, while also continuing to use WiFi even when signal quality has degraded significantly.

Additionally, mobile operating systems fail to appropriately provide the flexibility and diversity of resource management these devices require given the conditions and environments under which these devices operate. Unlike stationary hosts such as desktops and servers, mobile devices are often limited on several axes: battery life, network data usage, and more. Also, not all users have the same preferences; resource usage is dependent on many factors including network and workload, and may be required to be managed over epochs of varying lengths (e.g., hours for battery, days/weeks for network data). Furthermore, adding improved mobility to the network stack makes this issue even more pertinent. Being able to move between networks or use multiple ones simultaneously will require management to make sure limited resources are not spent wastefully.

1.1 Mobility and the Network

Incorporating mobility and multihoming into a network stack not designed for either has been difficult. Some attempts [32, 12] try to retrofit mobility into single transport protocols like TCP. However, due to constraints of working within the confines of the existing protocol, they produce subtle failure cases. Additionally, the work is not extensible to other transport protocols. Other mobility solutions do not work end-to-end [27, 42, 10], but rather redirect traffic through middleboxes (like home agents in Mobile IP), requiring network support and potentially inefficient “triangle routing.” While these network layer solutions require minimal changes to end-hosts, they interact poorly with existing transport protocols (e.g., TCP cannot distinguish congestion from loss during mobility) and do not have proper support for multihoming (e.g., individual data flows cannot migrate between network interfaces). Other solutions,

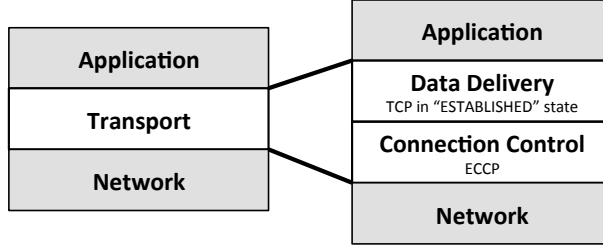


Figure 1.1: The TCP/IP transport layer broken up into two sublayers, with ECCP operating below data delivery.

like placing multiple wireless access points in the same virtual LAN (VLAN), support limited mobility only in certain cases; this solution does not let a device switch between different network providers (e.g., between cellular and WiFi).

In this thesis, we argue for an transport-agnostic, end-to-end solution that changes the end-host network stack. The network stack should support *path multiplicity* (where a single connection may be spread over multiple interfaces or paths) and *location dynamism* (where hosts can change locations without breaking ongoing connections) for **all** transport protocols, i.e., TCP, UDP, etc. To accomplish this, we argue for separating out the functionality of the current transport layer into two sublayers: (i) *connection control* (e.g., starting and stopping connections and their constituent flows, and changing their associated addresses) and (ii) *data delivery* functionality (e.g., best-effort versus reliable delivery, congestion control, flow control, etc), as shown in Figure 1.1. Decoupling the functionality of these two sublayers allows us to engineer a new end-to-end connection control protocol (ECCP) that sits above the network layer and works for multiple data delivery protocols.

The sublayer functionality offered by ECCP is markedly different from typical “layer 3.5” designs, like HIP [23] and LISP [10], which define new host/endpoint name layers to hide address changes from the stack rather than actively dealing with them through connection control. ECCP does not need invariant host/endpoint names; once a connection is established hosts can then, using end-to-end connection control, track changes to their addresses on-the-fly and inform all of their peer hosts

of the changes. Path multiplicity is supported by allowing a single connection to consist of one or more *flows*, each associated with an interface or path, similar to MPTCP [38]. Unlike MPTCP, however, the base functionality is reusable by any data delivery protocol, and flows can further change interfaces or addresses over time, without breaking the associated connection. Most existing solutions for mobility either do not handle path multiplicity or work on a per-interface instead of a per-flow basis, limiting the flexibility with which multipath and multihoming protocols can respond to mobility. Conversely, ECCP allows for the migration of individual flows independently, allowing better load balancing and more expressive policy for different types of flows.

Yet end-to-end control protocols like ECCP are notoriously hard to get right. This is exacerbated by subtle corner cases inherent to communication in an unreliable medium and the dynamism caused by device mobility and VM migration. In Chapter 3, we look at requirements for designing such a protocol including evaluation of other approaches, as well as the ECCP protocol in-depth, including how separating connection control and data delivery actually simplifies the design, engineering, and verification of the protocol.

From ECCP we were able to create a detailed state-transition diagram for the protocol, which guarantees that connectivity is preserved in the face of location dynamism and path multiplicity. This model and state diagram also formed the foundation of our implementation of ECCP in a larger network architecture called Serval [24]. Serval is presented in §3.4 and using a Serval prototype, ECCP is evaluated in Chapter 4.

1.2 Device Resource Management

Even though resource management is a principle task of operating systems (OSes), mobile OSes have been slow to tackle the unique resource challenges facing mobile

devices. These devices are restricted by a myriad factors, including energy, computing, and network data limits. Utilizing the network tends to trade off these (usually) limited resources. Some of these—such as battery life or monthly network data limits—must be managed over epochs many orders of magnitude greater than those relevant to traditional packet scheduling. The mismanagement of the network can be costly and provide subpar performance.

Unfortunately, today’s mobile OSes offer only a limited set of mechanisms to manage resources across users, operating conditions and longer time horizons, typically in the form of hard limits. Some OSes, e.g., Apple’s iOS and Windows Phone, limit which apps can run in the background and their network usage to improve battery life. Others, like the default Android OS, support setting simple hard data limits on the device; some manufacturers also introduce power-saving modes [1] that aggressively shut off background tasks and network usage, even if it hurts app functionality. Apps may also include additional settings to manage resource usage. These approaches lack sufficient flexibility and ease-of-use, as they are either too restrictive or require users to constantly micro-manage their device.

Even if third-party applications are well behaved and expose options to manage network usage, an application’s behavior may not always align with its user’s interests. Apps commonly focus on high performance to ensure a good user experience, rather than minimizing network usage and without concern of their impact on concurrently running apps. On the other hand, while some apps allow users to disable cellular usage or downgrade to lower bitrates, these degraded user experiences may be unnecessary given a device’s current data usage.

In Chapter 5, we introduce Tango, a system for managing network usage via a flexible, programmatic policy model, designed to simplify and better align a user’s interests with their device’s network management. Tango centralizes network management in a controller process, which handles monitoring device state, soliciting

interested parties (users and apps) for their preferences (encoded as programs called *policies*), and adjusting usage as applicable. This controller has user preferences codified into a single user policy, and acts as the mediator between user and app interests.

A policy is a program that uses the current device state (and for apps, usage constraints) as input, and returns a list of actions (e.g., move connection X to network Y, rate limit connection X to 500kbps, etc) to be executed on its behalf as output. Because of the level of dynamism in mobile devices, the device state is comprised of metrics for various sources including the kernel, network, and battery. For example, if the current state is the device is connected to a cellular network, a policy could then ask for a rate limit to be set in order to save data against a cap. But a policy can be even more expressive: when deciding on a network, a policy can consider the load on the network (e.g., number of apps using it), any priority (e.g., foreground) usage, and more. Thus, Tango’s model is richer and more flexible than what is currently available.

We recognize that these devices ultimately belong to the user, so their preferences must trump those of any app. Yet, we also recognize that apps can improve their resource usage in valuable ways when given guidance by the user. Tango’s approach to conflict resolution is *proactive*. That is, the user policy states upfront *constraints* on usage for apps. This allows app policies to be helpful and offer viable actions to the controller to execute on their behalf, while uncooperative or legacy apps simply are subjected to the constraints generated. *Reactive* policy conflict, such as changing or ignoring invalid policies, would limit how effective apps could be in optimizing usage since they do not know what bounds to stay within and whether their actions are being executed. Further, Tango allows app policies to “hint” about their desired usage, which user policies can use as part of subsequent evaluations.

Taken together, ECCP (and by extension Serval) and Tango provide new abstractions for addressing shortcomings in today’s devices when it comes to mobility. In

Chapter 2, we first provide a look at the existing solutions for dealing with network mobility and resource management. Chapter 3 then introduce our mobility protocol ECCP and its place in the Serval architecture, followed by evaluation of the two in Chapter 4. In Chapter 5 then introduces our resource management platform Tango with evaluation of our prototype in several case studies in Chapter 6. The thesis concludes in Chapter 7 with directions for future work and contributions.

Chapter 2

Background

In this chapter we look at the problems—and attempted solutions—with using mobile devices today caused by a network stack that does not account for mobility and operating systems that do not offer users enough flexibility to effectively manage their resources.

2.1 Supporting Network Mobility

Mobile devices, e.g., smartphones and tablets, have made network mobility a common occurrence, especially compared to previous computing devices, e.g., desktops, servers, and mainframes. Today, devices are not always connected to a single network, but rather they can connect to multiple networks as a user commutes from work to home, walking from their dorm to their class, or strolling down a city street past WiFi hotspots at every coffee shop. Meanwhile, the user is trying to browse the web, stream music, or watch the latest viral video. The last thing they want is for the connection to break and their apps to stop working while the device tries to reconnect.

Yet that is what happens with today’s network stack. The network stack currently used by mobile devices is a poor match for mobility due to overloading the meaning of

addresses (to identify interfaces, demultiplex packets, and identify sockets) and port numbers (to demultiplex packets, differentiate service end-points, and identify application protocols). Traditionally network connections are identified and demultiplexed on a “five-tuple” consisting of the protocol (e.g., TCP, UDP), source IP and port, and destination IP and port. Unfortunately, by doing this each endpoint is identified its port and IP, which prevents it from changing networks (where it would get a new, different IP).

To overcome this, previous solutions can be roughly divided into two broad categories. The first is work that provides a transport-protocol agnostic solution at the network layer (or below), and the second is work that aim to support such functionality in specific transport protocols.

The canonical approach for providing mobility regardless of transport protocol is to rely on encapsulation [27, 28, 23, 42, 10], where packets carry two pairs of addresses; a pair of unchanging (invariant) addresses that identifies the host/endpoint of the connection, and a location-dependent address pair. The invariant addresses identify peers in the network and facilitate demultiplexing across location dynamism, while the other address pair directs packets across the network. The main differences between each encapsulation scheme lie in how they initially setup encapsulation and how they signal changes when hosts move or migrate flows between interfaces. For instance, HIP [23] uses an *end-to-end* protocol to setup encapsulation, while LISP [10] and Mobile IP [27] rely on *in-network* infrastructure to handle this in a more transparent way to the endpoints.

By separating the addresses that are used for identifying the connection and locating the endpoints, the locations are able to change while connection is maintained. However, these solutions are not without their downsides. Using encapsulation adds an extra layer of complexity to the problem, and these methods of indirection can have performance implications. For instance, Mobile IP relies on an in-network infrastruc-

ture solution that introduces a “home agent” that tracks the mobile client and acts as a proxy. Because of this, Mobile IP uses “triangle-routing”: packets from a mobile client go directly to its destination, but packets on the reverse path go through the home agent (incurring extra latency) in order to reach the mobile client. Given that mobile clients are becoming the norm, supporting mobility should be something the network stack provides natively, not something added on via encapsulation, especially if it comes at a performance cost.

Other prior work has taken a different approach, instead providing mobility support by modifying individual transport protocols [32, 12, 38]. These solutions extend the transport protocol’s (typically TCP) signaling to handle address changes, which allows the transport protocol to be aware of mobility events. In contrast, encapsulation schemes in general handle signaling in another layer (or outside the stack altogether), which leaves the transport protocol to recover on its own during mobility events. This can have detrimental effects on performance. For instance, TCP would confuse mobility events for spikes in round-trip times or a significant drop in bandwidth and incorrectly adjust its retransmission timers and congestion windows. Modifying the transport protocol allows for those protocols to deal with those cases more gracefully, but with one obvious downside: only adding support for mobility to one protocol. The solutions also tend to repurpose parts of the protocol (e.g., TCP options) in ways that can be brittle if other modifications are in place.

Additionally, while several of these transport modifications have been proposed, these solutions can have unacceptable results (Chapter 3), including losing connectivity. For example, protocols like TCP Migrate [32] and HIP [23] can get “stuck” (i.e., not be able to continue sending data) when a client migrates between networks multiple times in quick succession; for other solutions, such as those that require in-network middleboxes [27, 10, 42], rapid migration may not even be possible. This is unacceptable for today’s devices, which are often used in environments (e.g., city

blocks or college campuses) with several hotspots—with fluctuating signal levels—to choose from. Attempting to use the best one or falling back to cellular can cause multiple network switches to occur quickly. Another protocol based on TCP, Multipath TCP (MPTCP) [38] defines a modification that can stripe a data stream across multiple TCP subflows, using different network paths. MPTCP supports mobility by simply starting additional subflows on new addresses, tearing down subflows on obsolete ones. While this masks changes in connectivity (and used in limited capacity on Apple’s iOS [34], it also imposes performance penalties because each new flow requires establishing new state and re-entering TCP slow start.

In summary, while there are solutions for mobility today, they are not without their problems. There are performance implications either by introducing new layers of complexity/encapsulation, forcing traffic through inefficient routes, or confusing other protocols that are not aware of the added mobility functionality (e.g., TCP not being aware that something is a migration event rather than a connection problem). Further, for a feature as common as mobility, many of these solutions introduce it as an add-on or afterthought to legacy protocols. Instead, mobility should be performant, always correct, and integrated with the network stack.

2.2 Handling Limited Resources

Mobile devices offer more flexibility to users, but also force users into making trade-offs while using them, e.g., connectivity vs data caps or performance vs battery life. Also, while adding network mobility could address some pain points of mobile devices, it would still need a framework to manage when to use different networks and how much usage to put on that network. Since mismanaging network costs can turn into real world costs (e.g., overage charges with regards to data caps), users should be able to express their preferences and have the device manage that for them.

Even without seamless network mobility, there has been work on deciding which network to use, given the rise of multi-homed mobile devices. These works use a variety of approaches, from rule-based decision engines [37, 41] to evaluating based on different utility functions [25], these systems attempt to decide which network is best given the current workload. However, these systems often bake in assumptions about what the user wants and what they are using the system for, offering little room for different preferences. Further, network choice alone is only one part of the resource management problem. It is fairly easy to pick the right network to stay connected, but end up misusing it to burn through a data cap or battery (Chapter 6). Using the network responsibly so as not unnecessarily run up data costs or drain battery is important as well.

As mobile device platforms—particularly smartphone operating systems (OS)—have matured, manufacturers, app creators, and those building the OS have recognized the need for tools to better control resource usage. Some platforms integrate limitations into the platform itself, by limiting the capabilities of third-party apps, e.g., which apps can run in the background. The goal of these limitations are usually to preserve the most limited of resources such as battery life and cellular data use [16, 18]. Long-running background usage is only available to apps in certain classes (e.g., playing audio), otherwise apps can hand off short-lived tasks (e.g., finishing a download) to the OS to finish. Such restrictions may be useful for lengthening a device’s battery life, but takes away control from the user. A user who has plentiful charging opportunities and WiFi would find these restrictions unnecessary and limits what they can do with their device. Ideally, a resource management solution would not have these artificial limits.

Other platforms, notably Android OS, do not impose such restrictions, but offer other mechanisms to manage resource usage. By default, Android includes tools like a Data Usage settings pane for managing how much usage “metered” (e.g., cellu-

lar) networks are allowed. Recent versions and other manufacturers have introduced special modes to conserve battery, e.g., Sony’s STAMINA mode [1] or the Doze [17] feature in Android 6.0. A user can opt-in to use these modes, which determine when a phone is no longer in active use and then limit network and CPU usage. Further, app creators sometimes take it upon themselves to include settings to help manage their usage (e.g., only stream music on WiFi). All of these options are helpful, but have two main downsides. First, they are typically rather limited, offering “on or off” behavior or only a handful of levels. A user may be fine with limited cellular usage, but it can be tough to decide what that means, e.g., “low” versus “medium” quality for media streaming. Second, getting all these settings right can be a management nightmare for users (Chapter 5). Users not only have to monitor that the usage is actually fitting with their preferences—by checking several system menus—but also keep up to date if apps change, remove, or add new settings to configure.

Other work has been proposed to specifically tackle arguably the two most limited resources on mobile—battery and network data—by opportunistically using the cheapest network (usually WiFi). Several systems [3, 22, 30] try to use past data and network conditions to make predictions about whether a network is useful for saving either data or battery. These systems batch network requests while on cellular and wait for WiFi if possible, usually having some upper limit on the amount of delay. These strategies are certainly useful, but only for certain kinds of traffic (e.g., email, background syncing). How reliably they could classify traffic and their effectiveness from other app types (e.g., browsing, streaming, etc) is unknown.

The network resource management problem for mobile devices is certainly multifaceted, with different parties (e.g., users, platform owners, app creators) attempting to influence the process. Context information from the device is starting to be useful to influence resource management (e.g., Doze, STAMINA, the WiFi offloading works), but those tools also lack sufficient flexibility without micromanaging. The situation is

improving, but there is still more to be done to make network resource management flexible and effective.

With these problems and attempted solutions in mind, the following chapters present our approaches for improving mobility and managing network resources in today's devices.

Chapter 3

Supporting Mobility

As we have seen in the previous chapters, the need for network mobility has become paramount. Multiple network technologies (e.g., 4G and WiFi) and multiple path choices per technology (e.g., WiFi access points) offer choice, if only the devices could take advantage of it. While changes could be made to the network to facilitate this [27], it is not strictly necessary. Instead, reorganizing parts of the network stack to better delineate responsibilities and remove overloading terms can make this possible on end-hosts themselves. In this chapter we introduce ECCP, a protocol designed to do that, as well as Serval, a network architecture redesign that uses ECCP as a key component.

First, in §3.1, we introduce the goals of the protocol and related efforts. §3.2 describes our design of ECCP, including how we address mobility and multihoming. §3.3 covers a few extra aspects of the protocol including its formal verification, security, and extensions. Finally, §3.4 introduces Serval which uses ECCP and will serve as an evaluation platform in Chapter 4.

3.1 Protocol Requirements and Related Work

In this section, we define requirements to be met by an end-to-end connection control protocol to correctly handle both location dynamism and path multiplicity. We also discuss past works and why they do not meet our requirements.

3.1.1 Protocol Requirements

In today’s network stack, the transport layer is responsible for establishing a connection to another endpoint, and then taking an application stream and dividing it into packets to send over the connection. On the receiving side, the transport layer demultiplexes packets based on a five tuple (IP addresses, ports and protocol number) and reassembles the application stream (while correcting for packet loss and reordering when necessary, e.g., for TCP). This conflates data delivery and connection control functionality. In this work, we treat connection control and data delivery as logically separate, focusing on the requirements of connection control.

Traditionally, connection control happens at the *beginning* and the *end* of a connection, i.e., when establishing and tearing down a flow. However, to support mobility, connection control should fulfill the following requirement: *two communicating hosts are guaranteed continued connectivity even when the network addresses of either host changes or some (but not all) of its interfaces go down.*

Support for this requirement is frustrated today by the overloading of IP addresses to indicate both the location and identity of a host. This overloading means that every location change is *also* an identity change, which leads to broken connections whenever hosts move and change their addresses or move flows to a different interface. A change in address (i) invalidates the five tuple used to identify the communication context in the network stack, and (ii) obsoletes the address used by a remote endpoint to send packets. A mobility solution must address both these issues; i.e., ensure that

Feature	ECCP	HIP	Mobile IP	LISP	ROAM	MPTCP	TCP-Migr	TCP-R
Formally verified	yes	incorrect	no	no	no	no	incorrect	no
Per-flow migration	yes	no	no	no	no	no	yes	yes
Rapid migration	yes	no	no	no	no	n/a	no	no
End-to-end	yes	yes	no	no	no	yes	yes	yes
Multipath capable	yes	yes	no	no	no	yes	no	no
Trans. protocol agnostic	yes	yes	yes	yes	yes	no	no	no
Avoids encapsulation	yes	no	no	no	no	yes	yes	yes

Table 3.1: Comparison of ECCP with alternative approaches

the demultiplexing key remains valid and that all communication peers are informed of address changes, even during frequent mobility and migration. We develop a connection control protocol that allows hosts to signal address changes in the *middle* of a connection. This protocol must operate correctly even when control packets are lost, reordered, duplicated, or arbitrarily delayed, and must ensure connectivity in both directions.

It is important to note that no end-to-end signaling protocol can handle the case of *simultaneous movement*, as rare as that may be. However, the system as a whole should ensure continued connectivity in such a case; we propose a simple, lightweight in-network mechanism to handle this special case in §3.3.3.

3.1.2 Related Work

We divide prior work on protocols for location dynamism and path multiplicity into two classes: (i) those that provide a transport-protocol agnostic solution at the network layer (or below), and (ii) those that aim to support such functionality in specific transport protocols. Table 3.1 gives an overview of the most relevant prior works, along with correctness properties and features, as discussed below.

As covered in the previous chapter, for a transport-agnostic solution, much of the work has focused on using encapsulation [27, 28, 23, 42, 10]. Packets will have two pairs of addresses, where one pair identifies the connection endpoints and never changes and the other is a pair of (possibly-changing) location-dependent addresses. Most differences between them are between how they setup encapsulation and how

they signal changes in host locations, e.g., HIP [23] using an end-to-end protocol to setup encapsulation versus LISP [10] and Mobile IP [27] using in-network infrastructure.

In contrast, ECCP does not rely on encapsulation, invariant host/endpoint identifiers, or in-network support. Instead, ECCP’s in-stack *end-to-end* connection control only requires an up-to-date address of a peer to initiate communication (typically acquired through DNS or some alternative service resolution mechanism [24]). Once communication state has been established, ECCP assigns this state a local ephemeral identifier (a flowID), which is used to signal changes in addresses on a per-flow basis. As a result, ECCP does not rely on semantically overloaded IP addresses and ports for demultiplexing, thus sidestepping the five-tuple issue. While ECCP addresses the identifier overloading in the stack, the downside is that both endpoints must be modified, which is not the case for most encapsulation schemes (HIP being the exception). On the other hand, ECCP requires no network support and simplifies the implementation of transport protocols. Further, ECCP has proper multihoming support by allowing per-flow migration between network interfaces, while encapsulation moves all flows associated with a particular host identifier, giving less fine-grain control over which interface a particular data flow uses.

In addition to the numerous encapsulation schemes, a number of prior works aim to provide mobility support by modifying individual transport protocols [32, 12, 38]. These solutions typically extend the transport protocol’s signaling to handle address changes, but doing so in a backwards compatible way can lead to incorrect results, as discussed below. In contrast, encapsulation schemes in general handle signaling outside the endpoint stack, which leaves the transport protocol to recover on its own during mobility events. This can have detrimental effects on performance, e.g., if TCP is in a long retransmission timeout. Due to ECCP’s new division of labor in the network stack, it has both transport-agnostic signaling and good integration

across multiple transport protocols, allowing retransmission timers to be frozen during mobility events. Ford and Iyengar [11] have proposed a similar division of labor, although not for the purpose of mobility.

TCP-R [12] was the first proposal for a modification to TCP to handle mobility, but did not offer any details about protocol operation such as sequencing or retransmission. TCP Migrate [32] specified a protocol for migration by allowing IP addresses to change, similar to ECCP. However, as shown in §3.2.3, TCP Migrate has misbehaving corner cases that can cause incorrect behavior during rapid migrations (for instance, moving a flow from one interface to another and back in quick succession). Other end-to-end signaling solutions, like HIP [23], share similar problems by relying on sequence numbers instead of version numbers for address updates, forcing the migration protocol to wait for out-of-date migration updates that may never arrive (§3.2.2). Protocols that rely on in-network middleboxes for migration [27, 10, 42] do not suffer incorrect behavior during rapid migrations (due to signaling/forwarding through a fixed rendezvous point), but are not “rapid” due to slower updates. In contrast, ECCP correctly supports rapid flow migrations in one round trip, by using an in-stack control protocol with version numbers.

Multipath TCP (MPTCP) [38] defines a modification to TCP that can stripe a data stream across multiple TCP subflows, using different network paths. MPTCP supports mobility by simply starting additional subflows on new addresses, tearing down subflows on obsolete ones. Although ECCP shares similarities with MPTCP’s control protocol, ECCP does not rely on TCP options. Instead, ECCP defines a new end-to-end control protocol underneath the transport layer, thus benefiting multiple transports. Hence, MPTCP’s data delivery functionality, such as congestion control and data segmentation, could run on top of ECCP.

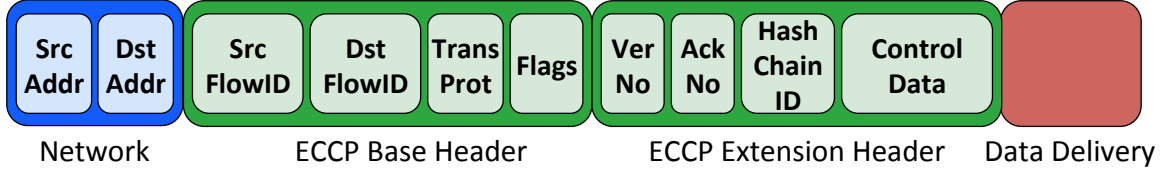


Figure 3.2: A conceptual view of a packet, showing the location and composition of ECCP headers. The data delivery header is typically a regular transport header (e.g., TCP), although used only to provide data delivery functionality.

chine, shown in Figure 3.1. In this section, we describe the protocol and highlight the design decisions we made.

3.2.1 Establishing a New Connection With a Single Flow

ECCP establishes connections and their constituent flows, and creates the state necessary to map between flows and the underlying interfaces used for transmission. An established connection needs to demultiplex packets to flows and be robust to mobility events.

Decoupling demultiplexing keys from addresses. Each flow is assigned its own identifier, called a *flowID*, which is essentially an opaque demultiplexing key that maps packets to socket state. *The usage of flowIDs avoids coupling demultiplexing with specific addresses*, as in the traditional “five tuple”, so that mobility does not affect demultiplexing. FlowIDs are put in an ECCP header in-between the network and data delivery headers, as shown in Figure 3.2. All data packets must carry at least the ECCP base header in order for the receiving endpoint to be able to demultiplex the packet, while ECCP control packets need not carry data. Note that the flowIDs replace the transport header ports for the purpose of demultiplexing, except for the first SYN packet when the destination flowID is not yet known, as we explain below.

Using separate demultiplexing keys on each host. ECCP uses explicit flowIDs that uniquely identify the flow. Each flow has two flowIDs, one for each host, rather than a single shared identifier. Each host demultiplexes incoming packets

Abstraction	State
Connection	local connection version #, remote connection version # list of flows, remote interface list (IList)
Flow	local flowID, remote flowID local flow version #, remote flow version # local Address, remote Address

Table 3.2: State stored by ECCP for connections and flows

using only its local flowID, but includes the remote flowID in outgoing packets so the receiving host can demultiplex on its own identifier. This allows hosts to change their own flowIDs when migrating, which is useful for NATs [2] and ensures the uniqueness of the demultiplexing key on each host.

Exchanging alternate interface addresses for connection resilience. During connection establishment, the communicating end-hosts exchange a list of peer interfaces (IList) that can be used for establishing new flows. ILists are placed in an ECCP extension header and increase connection resilience by enabling flow establishment on alternative interfaces if the interfaces used by active flows become unavailable.

Confirming reverse connectivity. Network paths can exhibit asymmetric connectivity, where host A can reach B but B cannot reach A. In ECCP, we only allow connections along paths on which each host is able to reach its peer; its three-way synchronization handshake confirms reverse connectivity with its final acknowledgment (ACK). Like TCP, this handshake protocol is used during connection establishment; additionally, the handshake protocol is used when an established flow changes addresses (which TCP does not support), as discussed in §3.2.3.

Connection establishment is shown in Figure 3.3. This handshake initializes the state of the connection and a single initial flow, as enumerated in Table 3.2. The first SYN packet does not carry a known destination flowID, so this packet is demultiplexed to a listening socket based on its service, typically represented by a port number or

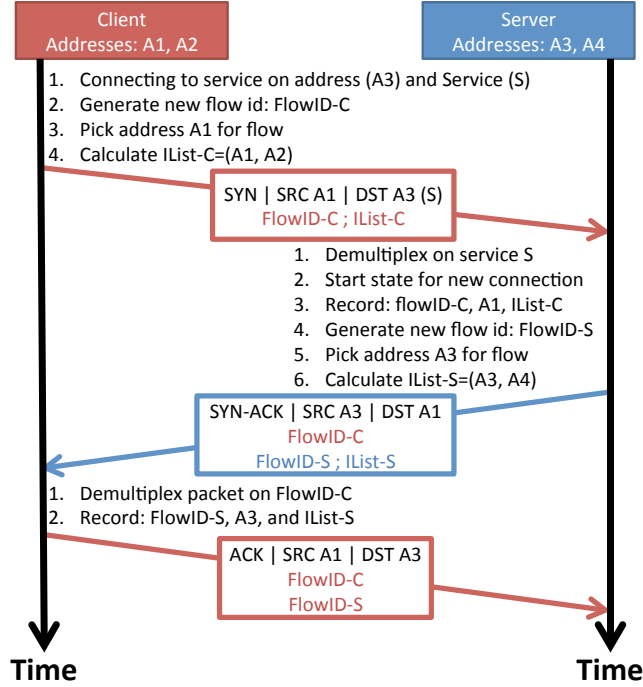


Figure 3.3: The ECCP protocol for establishing a new connection, using a three-way handshake similar to TCP. However, ECCP endpoints also share which interfaces they allow flows on and use flowIDs instead of IP/port.

other service identifier [24] carried in the packet. After establishing a connection, ECCP places the appropriate IP addresses and flowIDs in outgoing packets, and future packets are demultiplexed based on the destination flowID only.

3.2.2 Adding Flows to an Existing Connection

Either endpoint can add flows to an existing connection, in order to spread traffic over multiple interfaces or paths. Figure 3.4 shows how a client adds a flow between local address A2 and server address A4; the steps for the server to add a flow are analogous.

Supporting flexible policies for interface selection. To establish a new flow, the two endpoints must agree on which pair of interfaces to use. Each host may have its own policies for selecting interfaces, based on performance, reliability, and cost. For example, a smartphone user may prefer to use a low-cost, high-performing WiFi

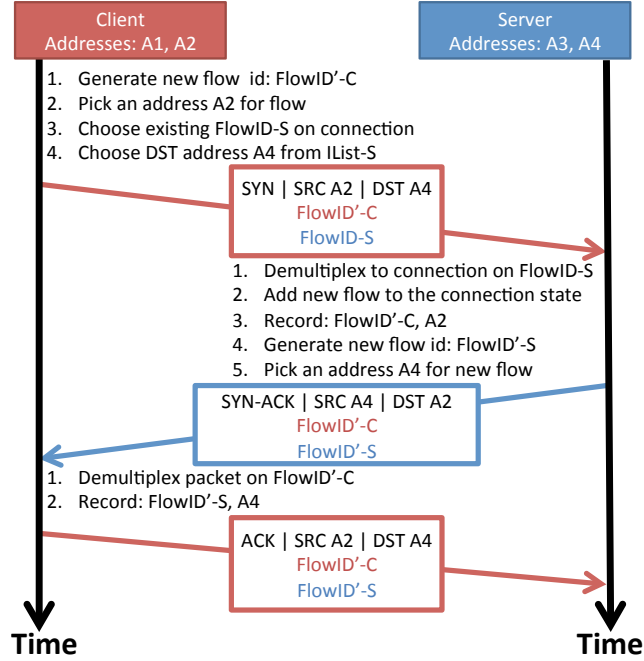


Figure 3.4: Adding a new flow to an existing connection in ECCP.

interface for high-bandwidth applications, instead of the more reliable (but more expensive) cellular interface. (If the WiFi connectivity is no longer available, the endpoint could migrate the flow to the cellular interface to continue the connection.)

To support flexible local policies, ECCP allows each endpoint to select its own interface. The initiating host selects a local interface (and associated IP address) for the new flow, and sends a SYN packet to one of the interfaces in the IList of the remote endpoint. Upon receiving the SYN, the remote endpoint either agrees to establish a new flow on the interface it received the SYN packet on, or it responds with a NACK packet, as shown in Figure 3.5. An alternative to using a NACK would be for the peer to simply respond with a SYN-ACK from the interface it prefers to use, but this approach fails to test connectivity from the initiating host to the preferred interface prior to establishing the connection. Note that while the initiating endpoint may influence the decision (e.g., by picking a remote interface based on past performance), the remote endpoint has the final say on which of its local interfaces to use.

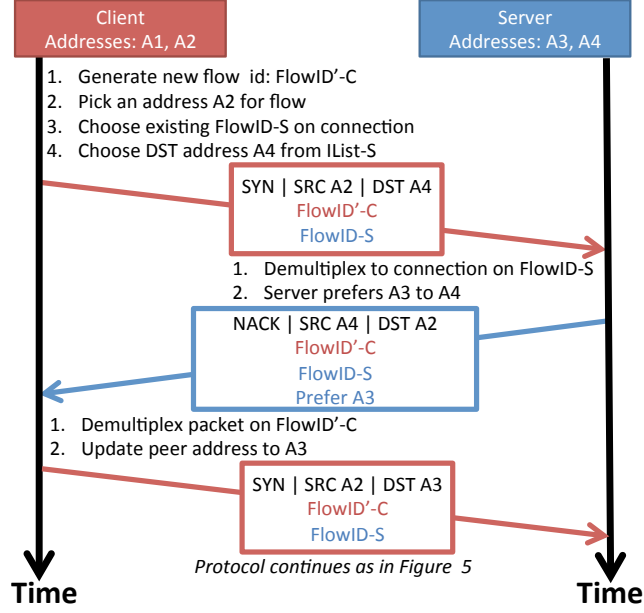


Figure 3.5: Choosing a different interface for a new flow in ECCP.

3.2.3 Changing the IP Addresses of Existing Flows

When a host changes location due to device mobility, VM migration, or failover, it needs to preserve flow connectivity by notifying its peers of its new network address(es). We present the ReSYNchronize protocol used to update the peers in Figure 3.6, where the mobile host changes its address and notifies the stationary host. The mobile host can optionally change its own flowID during migration (see section on NATs in [2]). Once a mobile host establishes a new address for one of its interfaces, it runs this protocol on every flow using that interface.

Protocol for updating ILists. This resynchronization protocol is also used to update the IList, even if the address on active flows does not change (e.g., an alternative interface established connectivity). In that case, the new address on the flow simply remains the same as the old one; only the IList changes. The IList is always updated as a single entity with the new list overriding the old one. No incremental update protocol is provided, in order to avoid introducing convergence issues where the two communicating end-hosts disagree on the contents of the IList.

Because the IList is not very large, the amount of communication overhead saved with an incremental update protocol is not worth the added protocol complexity.

Use of version numbers. Upon receiving an RSYN packet, a host needs to determine that the change of address does not reflect a past event. For example, if a client moves from address A1 to A2 to A3, the server may receive the migration request for A3 before A2, and should therefore ignore the migration to A2 since it is no longer valid. To avoid acting on past events, ECCP uses *version numbers*, which are separate from any sequence numbers used by the data delivery protocol to, for example, implement a reliable data stream on top of ECCP. This allows ECCP to support different data delivery protocols. Version numbers semantics are also markedly different than the familiar semantics of TCP-style sequence numbers: while sequence numbers require processing all packets up to $N-1$ before processing packet N , ECCP's version numbers simply require that the packet being received has a greater version number than any packet seen previously. Any packets that arrive with a version number less than the largest version number a host has seen are ignored. These semantics are necessary for correctness: migration message processing should not be delayed waiting for stale migration messages, as they may not be deliverable. (And even if they are, stale information is not useful anyway, as the interface address has subsequently changed.) These semantics are also robust in the face of rapid migrations: previous migration attempts do not need to complete before later ones. Notably, protocols such as HIP [23], which use sequence numbers for migration messages, can break under rapid migration because a new migration may occur before an older migration has been acknowledged. Finally, ECCP employs two separate types of version numbers: (i) one for each *flow*, used to order the migration requests of each flow, and (ii) one for each *connection* that orders updates to the IList.

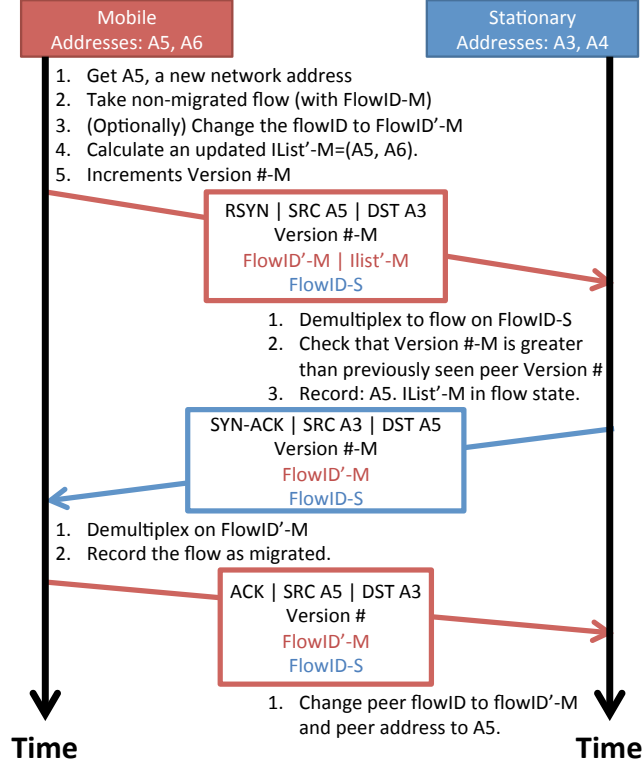


Figure 3.6: The ECCP protocol for changing the address associated with an already established flow. FlowId_m and FlowId_s are the IDs for the flow, while A5 is the new address and (A5,A6) is the new interface list.

Explicit acknowledgments. The ECCP protocol requires a migration acknowledgment to explicitly include the version number of the flow's migration. Alternatively, TCP Migrate uses the fact that it received data packets on the new address as a de facto acknowledgment that a migration message was received. We found this method of implicit acknowledgments to have incorrect corner-cases. We illustrate one such incorrect case in Figure 3.7, in which the packet sent at time T1 is delayed and received at time T3, where it is assumed (incorrectly) to acknowledge the packet sent at time T2 (which is lost). At the end of this trace, the stationary host believes that the mobile host is at address A12, while its real address is A11. At the same time, the mobile host believes that its migration protocol has completed successfully. In ECCP, ACK packets carry the version number of the flow migration and thus avoid this issue, but they can still be piggy-backed on data packets.

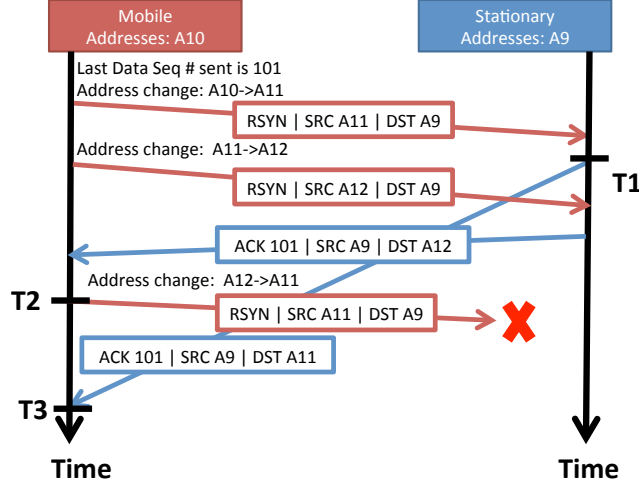


Figure 3.7: An example of a misbehaving protocol trace when using implicit ACKs to confirm migration.

3.3 Other Concerns

3.3.1 Verification

Because of the edge cases and general difficulty of designing a network protocol that supports migration (Table 3.1), the original ECCP work [2] formally verifies the protocol. While not directly related to the work of this thesis, it is nonetheless important since ECCP is used as a major feature of Serval, which in turn is used in our evaluation of Tango.

The formal verification explored in the original paper uses SPIN [14] to model the protocol. The formal verification checks the correctness of ECCP by checking that it is safe (i.e., free from deadlocks) and liveness. Using novel techniques to model an unreliable network and to reduce the search space that needs to be checked, we verified that ECCP was free from deadlocks and livelocks through at least 6 migrations. Further details about this process can be found in the original paper [2].

3.3.2 Security

ECCP, like other connection-based network protocols, is potentially vulnerable to two main classes of malicious attacks: denial of service (DoS) and hijacking. A protocol is particularly vulnerable to a DoS attack if a request from an unverified party can cause a host to spend an asymmetric amount of resources. The classic example of a DoS attack is SYN flooding, where cheaply crafted (and typically spoofed) SYN packets cause a server to allocate kernel memory buffers. Nothing in the ECCP protocol requires excessive memory or computation to process the initial handshake or the migration protocol. SYN cookies [5] can also be used to prevent the allocation of kernel state to a new connection before return reachability is tested.

Protocol support for migration introduces new potential threats from attackers, who may try to (i) hijack ongoing connections by inserting control messages into the communication stream, or (ii) disrupt connections by sending fake migration messages. Some protocols [32] rely on using public-key cryptography to secure control packets to reduce risks of both *off-path* and *on-path* attackers. However, this is typically a computationally expensive operation.

Instead, in ECCP, we use hash chains [20] to address hijacking attempts¹. When beginning a connection, both ends derive a nonce, M , and apply a one-way hash function for N iterations. N should be larger than the number of control messages expected in a connection (e.g., 100). Each side stores the chain locally and exchanges the final result when creating the connection. Each control message thereafter will then send an intermediate value of the hash chain that is $X-1$ iterations from the end result, where X is the version number of the control packet. This other side will perform the hash function for $X-1$ iterations to confirm that the final result is the initial value exchanged.

¹Originally in [2] we only used a nonce to prevent off-path attackers but not on-path attackers.

This will prevent both off-path attackers as well as on-path attackers who cannot modify packets but can inject their own packets. Neither type of attacker will be able to guess the correct intermediate hash value since the hash function is one way, so even having the end result (e.g., the on-path attacker) is not useful. These hash functions are typically very fast—the latest candidates for SHA-3 can do thousands of Mbps [31]—so they will be better than doing public-key cryptography. Further, the hash is only needed when verifying control messages and not data messages, which are comparatively infrequent (i.e., only on migration events).

3.3.3 Simultaneous Movement

Although not a usual occurrence, an ECCP connection is robust to simultaneous movement as long as both endpoints do not move before receiving the other endpoint’s address update (RSYN). In other words, the protocol can survive an incomplete three-way handshake and simply requires that *one* packet gets through to a peer before the peer itself can move. Note that no end-to-end signaling protocol can, by itself, handle the rare case when both hosts send address updates to each other and then simultaneously move before either receives the other’s update.

However, we can handle this rare case by adding an optional, lightweight redirection cache in the local network of either communicating host. This cache keeps short-lived redirection state pointing to the new locations of hosts that have recently migrated out of its network. The address of the in-network box responsible for the cache can be learned by a mobile host when joining the network (e.g., through DHCP). When a mobile host moves, it sends a message to the redirection box of its old network to add a pointer to its new location. Upon getting a new cache entry, the redirection box takes over the now migrated host’s old address for the duration of the cache entry (via gratuitous ARP-flooding or a similar mechanism). In this way, the redirection box will receive all messages meant for the migrating host, including the

RSYNs sent by peers that have moved simultaneously. Upon getting such an RSYN, the redirection box simply redirects the packet (through, e.g., encapsulation or address rewriting) toward the host’s new address. All packets other than RSYNs can be dropped by the redirection box. Upon getting a redirected RSYN, the migrated host learns the new address of its peer that moved simultaneously and initiates its own RSYN handshake targeting this address.²

Note that the redirection cache only needs to keep its cache entries for short durations (e.g., seconds), as it only needs to redirect a single RSYN to a migrated host. Further, it is sufficient that only one of the migrated hosts have a redirection cache in its old network for this approach to be effective.

This scheme’s use of ephemeral redirection also benefits privacy. While triangle routing solutions such as Mobile IP [27, 28] (see §3.1) require an authoritative middlebox for each client that sees the full history of a client’s movements, a host in our setting only needs to notify the redirection box of its last visited network.

3.4 Bigger Picture: Serval

The splitting of the transport layer makes ECCP useful not only for mobility of client devices, but also for backend services which are increasingly run in VMs or containers on platforms like Amazon Web Services [15]. To that end, we used ECCP as a key component of a network architecture called Serval, which was designed to be better suited to handle the multiplicity and dynamism seen in today’s networks. This includes being able to handle mobility (e.g., mobile clients, migrating backend services), but Serval also addresses naming and managing backend services. For this thesis, we focus on how the areas of Serval that make use of ECCP and help improve the mobility of devices. Our Serval prototype also serves as our evaluation platform

²The migrated host does not send an RSYN-ACK in response to a redirected RSYN, because such an RSYN did not take the direct path that the handshake aims to verify for bidirectional connectivity.

for ECCP through various case studies in the next chapter. The Serval prototype also lets us consider when and how to use mobility in real-world scenarios, which we utilize in Chapters 5 and 6. For more details on the Serval architecture, including aspects not directly related to mobility, see the original paper [24].

3.4.1 Naming Abstractions

Serval uses two naming abstractions to reduce the overloading of identifiers within and across layers of the network stack. First, a host-local identifier that is an implementation of the *flowID* abstraction from ECCP (and also uses the *flowID* moniker). The second is a *serviceID*, which are used to name groups of processes offering the same backend service. Using these two identifiers in upper layers of the network stack, instead of the traditional IP/port, improve mobility for clients and backend services and reduce unnecessary coupling of functionality to IP/port.

Explicit Host-Local Flow Naming

As previously discussed (§3.2), ECCP splits the transport layer into two layers: one controlling the connection management and the other for data-delivery (e.g., TCP in the ESTABLISHED state). In Serval, flowIDs are the identifiers used by ECCP to name a connection on each host, replacing the traditional five-tuple. Figure 3.8 shows what packet headers look like in Serval; you will notice it is the same as ECCP's headers (Figure 3.2) with slight modifications. Serval benefits from using ECCP in the following ways:

Network-layer oblivious: By forgoing the traditional five-tuple, Serval can identify flows without knowing the network-layer addressing scheme. This allows Serval to transparently support both IPv4 and IPv6, without the need to expose alternative APIs for each address family.

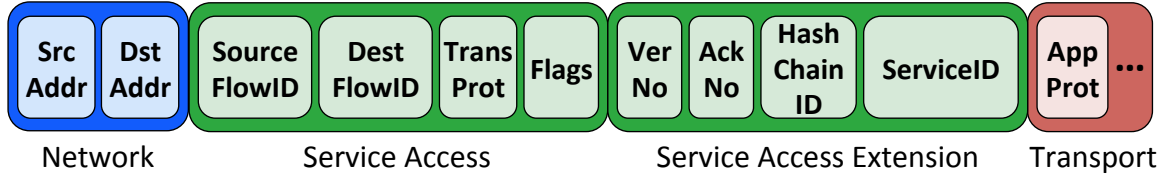


Figure 3.8: New Serval identifiers visible in packets, between the network and transport headers. Some additional header fields (e.g., checksum, length, etc.) are omitted for readability.

Mobility and multiple paths: FlowIDs help identify flows across a variety of dynamic events. Such events include flows being directed to alternate interfaces or the change of an interface’s address (even from IPv4 to IPv6, or vice versa), which may occur to either flow end-point. Serval can also associate multiple flows with each socket in order to stripe connections across multiple paths.

Middleboxes and NAT: FlowIDs help when interacting with middleboxes. For instance, a Serval-aware network-address translator (NAT) rewrites the local sender’s network address and flowID. But because the remote destination identifies a flow solely based on its own flowID, the Serval sender can migrate between NAT’d networks (or vice versa), and the destination host can still correctly demultiplex packets.

No transport port numbers: Unlike port numbers, flowIDs do not encode the application protocol; instead, application protocols are optionally specified in transport headers. This identifier particularly aids third-party networks and service-oblivious middleboxes, such as directing HTTP traffic to transparent web caches unfamiliar with the serviceID, while avoiding on-path deep-packet inspection. Application end-points are free to elide or misrepresent this identifier, however.

Format and security: The security benefits presented in §3.3.2 allow Serval to be protected from off-path attackers as well as certain on-path attacks.

Group-Based Service Naming

A Serval service name, called a serviceID, corresponds to a group of one or more (possibly changing) processes offering the same service (e.g., a web app, FTP, email, etc). ServiceIDs are carried in network packets, as illustrated in Figure 3.8. This allows for service-level routing and forwarding, enables late binding, and reduces the need for deep-packet inspection in load balancers and other middleboxes. A service instance listens on a serviceID for accepting incoming connections, without exposing addresses and ports to applications. When used in conjunction with ECCP’s concept of flowIDs, it efficiently solves issues of mobility and virtual hosting. While not directly related to ECCP, we briefly cover the benefits of serviceIDs here to give context to larger Serval design:

Service granularity: Service names do not dictate the granularity of service offered by the named group of processes. A serviceID could name services ranging from a single SSH daemon to an entire distributed web service. This group abstraction hides the service granularity from clients and gives service providers control over server selection. Individual instances of a service group that must be referenced directly should use a distinct serviceID (e.g., a sensor in a particular location). This allows Serval to forgo host identifiers entirely, avoiding an additional name space while still making it possible to pass references to third parties.

Format of serviceIDs: Ultimately, system designers and operators decide what functionality to name and what structure to encode into service names. For the federated Internet, however, we imagine the need for a congruent naming scheme. For flexibility, in our prototype we use a large 256-bit serviceID namespace, although other forms are possible (e.g., reusing the IPv6 format could allow reuse of its existing socket API). A large serviceID namespace is attractive because a central issuing authority (e.g., IANA) could allocate blocks of serviceIDs to different administrative entities, for scalable and authoritative service resolution. The block allocation ensures

that a service provider can be identified by a serviceID prefix, allowing aggregation and control over service resolution. The prefix is followed by a number of bits that the delegatee can further subdivide to build service-resolution hierarchies or provide security features. Other schemes are discussed further in [24].

Learning service names: Serval does not dictate how serviceIDs are learned. We envision that serviceIDs are sent or copied between applications, much like URIs. We purposefully do *not* specify how to map human-readable names to serviceIDs, to avoid the legal tussle over naming [8, 35]. Users may, based on their own trust relationships, turn to directory services (e.g., DNS), search engines, or social networks to resolve higher-level or human-readable names to serviceIDs, and services may advertise their serviceIDs via many such avenues.

3.4.2 The Serval Network Stack

We now introduce the Serval network stack, shown in Figure 3.9. The stack offers a clean service-level control/data plane split: the user-space *service controller* can manage service resolution based on policies, listen for service-related events, monitor service performance, and communicate with other controllers; the Service Access Layer (SAL), which is ECCP’s connection management layer with some extra features, provides a service-level data plane responsible for connecting to services through forwarding over *service tables*. Once connected, the SAL maps the new flow to its socket in the *flow table*, ensuring incoming packets can be demultiplexed. Using in-band signaling via ECCP, additional flows can be added to a connection and connectivity can be maintained across physical mobility and virtual migrations. Applications interact with the stack via *active sockets* that tie socket calls (e.g., **bind** and **connect**) directly to service-related events in the stack. These events cause updates to data-plane state and are also passed up to the control plane (which subsequently may use them to update resolution and registration systems).

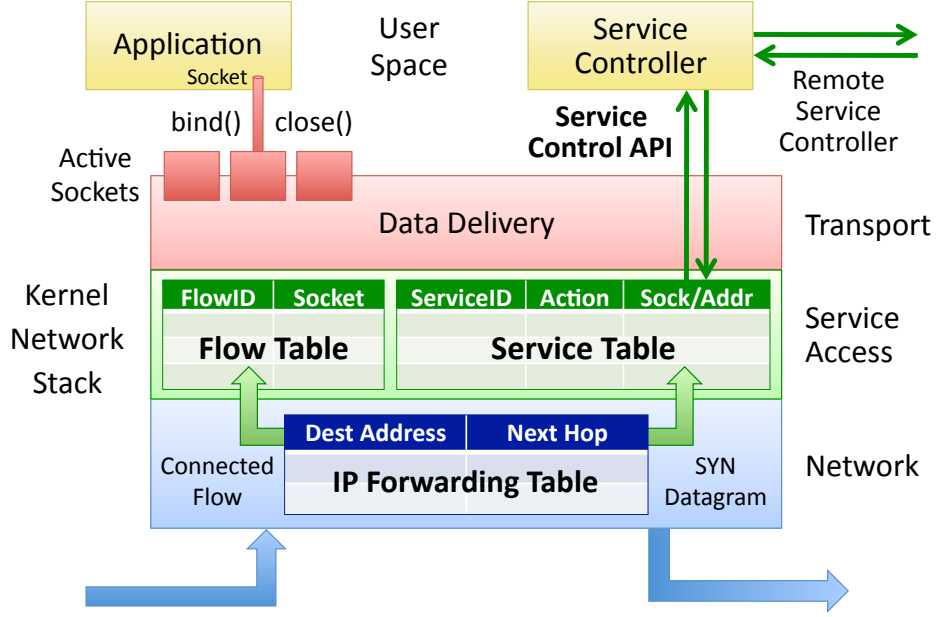


Figure 3.9: Serval network stack with service-level control/data plane split.

In the rest of this section, we first describe how applications interact with the stack through active sockets, and then continue with detailing the SAL. We end the section with describing how the SAL uses ECCP as an in-band signaling protocol.

Active Sockets

By communicating directly on serviceIDs, Serval increases the visibility into (and control over) services in the end-host stack. Through active sockets, stack events that influence service availability can be tied to a control framework that reconfigures the forwarding state, while retaining a familiar application interface.

Active sockets retain the standard BSD socket interface, and simply define a new `sockaddr` address family, as shown in Table 3.3. More importantly, Serval generates service-related events when applications invoke API calls. A serviceID is automatically *registered* on a call to `bind`, and *unregistered* on `close`, process termination, or timeout. Although such hooks could be added to today’s network stack, they would make little sense because the stack cannot distinguish one service from another. Because servers can `bind` on serviceID prefixes, they need not `listen` on

PF_INET	PF_SERVAL
<code>s = socket(PF_INET)</code> <code>bind(s,locIP:port)</code>	<code>s = socket(PF_SERVAL)</code> <code>bind(s,locSrvID)</code>
 <code>// Datagram:</code> <code>sendto(s,IP:port,data)</code>	 <code>// Unconnected datagram:</code> <code>sendto(s,srvID,data)</code>
 <code>// Stream:</code> <code>connect(s,IP:port)</code> <code>accept(s,&IP:port)</code> <code>send(s,data)</code>	 <code>// Connection:</code> <code>connect(s,srvID)</code> <code>accept(s,&srvID)</code> <code>send(s,data)</code>

Table 3.3: Comparison of BSD socket protocol families: INET sockets (e.g., TCP/IP) use both IP address and port number, while Serval simply uses a serviceID.

multiple sockets when they provide multiple services or serve content items named from a common prefix. While a new address family does require minimal changes to applications, porting applications is straightforward (§4.1.3), and a transport-level Serval translator can support unmodified applications (§4.3).

On a local service registration event, the stack updates the local service table and notifies the service controller, which may, in turn, notify upstream service controllers. Similarly, a local unregistration event triggers the removal of local rules and notification of the service controller. This eliminates the need for manual updates to name-resolution systems or load balancers, enabling faster failover. On the client, *resolving* a serviceID to network addresses is delegated to the SAL—applications just call the socket interface using serviceIDs and never see network addresses. This allows the stack to “late bind” to an address on a `connect` or `sendto` call, ensuring resolution is based on up-to-date information about the instances providing the service. By hiding addresses from applications, the stack can freely change addresses when either end-point moves, without disrupting ongoing connectivity.

A Service-Level Data Plane

The SAL is responsible for late binding connections to services and maintaining them across changes in network addresses. Packets enter the SAL via the network layer, or

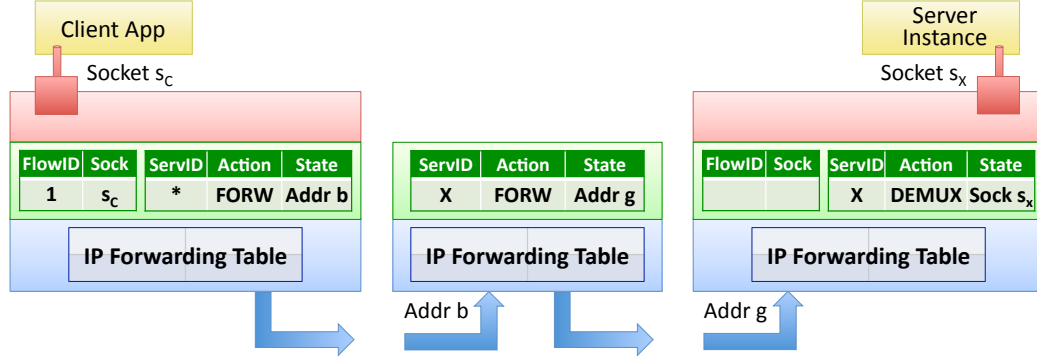


Figure 3.10: Serval forwarding between two end-points through an intermediate service router (SR).

as part of traffic generated by an application. The first packet of a new connection (or an unconnected datagram) includes a serviceID, as shown in the header in Figure 3.8. The stack performs *longest prefix matching* (LPM) on the serviceID to select a rule from the service table. ServiceID prefixes allow an online service provider to host multiple services (each with its own serviceID) with more scalable service discovery, or even use a prefix to represent different parts of the same service. More generally, the use of prefixes reduces the state and frequency of service routing updates towards the core of the network. The rules used by the service table are similar to an IP routing table, where packets can be forwarded (i.e., a FORWARD rule), sent to a local process (i.e., a DEMUX rule), and more [24].

Events at the service controller, or interface up/down events, trigger changes in service table rules. A “default” FORWARD rule, which matches any serviceID, is automatically installed when an interface comes up on a host (and removed when it goes down), pointing to the interface’s broadcast address. This rule can be used for “ad hoc” service communication on the local segment or for bootstrapping into a wider resolution network (e.g., by finding a local SR).

Figure 3.10 shows the use of the service table during connection establishment or connection-less datagram communication. A client application initiates communication on serviceID X , which is matched in the client’s service table to a next-hop

destination address. The address could be a local broadcast address (for ad hoc communication) or a unicast address (either the final destination or the next-hop SR, as illustrated). Upon receiving a packet, the SR looks up the serviceID in its own service table; given a FORWARD rule, it readdresses the packet to the selected address. At the ultimate destination, a DEMUX rule delivers the packet to the socket of the listening application.

Service-level anycast forwarding: Serval’s late-binding resolution and active sockets provide advantages for replicated backend services. A replicated backend service would have a SR that has rules for each instance of the service, and incoming SYN packets would first hit the SR, and then be balanced to the services to spread the load. Unlike current systems, however, the SYN packet is the only one that needs to go through this step, as future packets could simply communicate directly to their service instance. Since the SR lives near the service instances (e.g., the same datacenter), and its rules are updated via Serval’s active sockets, it knows when services go down and can direct new connections appropriately.

The indirection of the SYN may increase its delay, but the data packets, which make up the vast majority of the connection traffic, are unaffected. In comparison, the lack of indirection support in today’s stack requires putting load balancers on data paths, or tunneling all packets from the one location to another when hosts move.

End-Host Signaling for Multiple Flows and Migration

Serval utilizes ECCP as its in-band signaling protocol to support multiplicity and dynamism. With the SAL functioning as ECCP’s connection management layer, it can establish multiple flows (over different interfaces or paths) to a remote end-point, and seamlessly migrate flows over time. As mentioned in §3.4.1, Serval uses flowIDs instead of ports as a host-local demux key, and because we use ECCP, control mes-

sages (e.g., for creating and tearing down flows) are separate from the data stream and have their own message (i.e., version) numbers.

Multi-homing and multi-pathing: Serval can split a socket’s data stream across multiple flows established and maintained by the SAL on different paths. Consider the example in Figure 3.11, where two multi-homed hosts have a socket that consists of two flows. The first flow, created when the client C first connects to the server S , uses local interface $a1$ and flowID f_{C1} (and interface $a3$ and flowID f_{S1} on S). Using ECCP’s ILists, hosts can send a list of other available interfaces (e.g., $a2$ for C , and $a4$ for S) in a SAL extension header, to enable the other host to create additional flows using a similar three-way handshake. For example, if S ’s SYN-ACK packet piggybacks information about interface address $a4$, C could initiate a second flow from $a2$ to $a4$.

Connection affinity across migration: By ECCP’s design, the transport layer in Serval is unaware of flow identifiers and interface addresses. Therefore, the SAL can freely migrate a flow from one address, interface, or path to another. This allows Serval to support client mobility, interface failover, and virtual machine migration with a single simple flow-resynchronization primitive. Obviously, these changes would affect the round-trip time and available bandwidth between the two end-points, which, in turn, affect congestion control. Yet, this is no different to TCP than any other sudden change in path properties. Further, the SAL can notify transport protocols on migration events to ensure quick recovery. For example, a connection using TCP could temporarily freeze timers until the migration is complete to reduce the effect of lost packets during migrations on TCP state or TCP could re-enter slow start to more gracefully handle the path change.

Returning to Figure 3.11, suppose the interface with address $a3$ at server S fails. Then, then server’s stack can move the ongoing flow to another interface (e.g., the interface with address $a4$). To migrate the flow, S sends C an RSYN packet (for

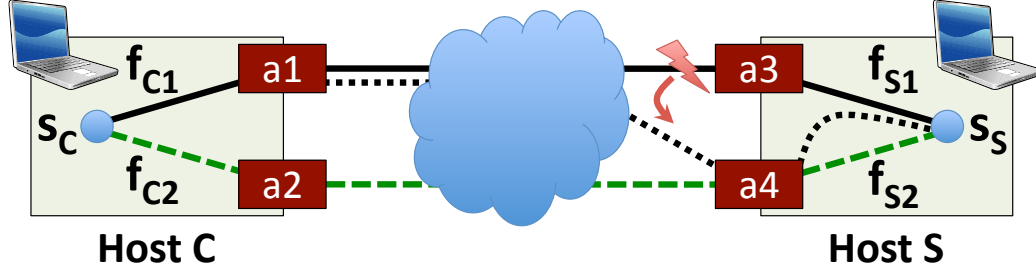


Figure 3.11: Schematic showing relationship between sockets, flowIDs, interfaces, addresses, and paths.

“resynchronize”) with flowIDs $\langle f_{S1}, f_{C1} \rangle$ and the new address $a4$. The client returns an RSYN-ACK, while waiting for a final acknowledgment to confirm the change. Sequence numbers in the resynchronization messages ensure that the remote end-points track changes in the identifiers correctly across multiple changes, even if RSYN and RSYN-ACK messages arrive out of order.

Additionally, since Serval is built on-top of ECCP, the stack has the same characteristics when it comes to simultaneous migration, security, NATs, etc (§3.3).

Chapter 4

ECCP/Serval Evaluation

4.1 Serval Prototype

For this thesis, we focus on evaluating ECCP as part of a Serval prototype and the insights gained from that experience. Through prototyping, we have (i) learned valuable lessons about our design, its performance, and scalability, (ii) explored incremental-deployment strategies, and (iii) ported applications to study how Serval’s (and ECCP’s) abstractions benefit them. In this section, we describe our Serval prototype and expand on these three aspects.

4.1.1 Lessons From the Serval Prototype

Our Serval stack consists of about 28,000 lines of C code, excluding support libraries, test applications, and daemons. The stack runs natively in the Linux kernel as a module, which can be loaded into an unmodified and running kernel. The module can run on Android, enabling us to easily bring connection migration to mobile devices.

The prototype supports most features—migration, SAL forwarding, etc.—with the notable exception of multi-path, which is left as future work. The SAL implements the service table (with FORWARD and DEMUX rules), service resolution, and end-

point signaling. The service controller interacts with the stack via a Netlink socket, installing service table rules and reacting on socket calls (`bind`, `connect`, etc.). The stack supports TCP and UDP equivalents, where UDP can operate in both connected mode (with service instance affinity) and unconnected mode (with every packet routed through the service table).

The introduction of the SAL inevitably had implications for the transport layer, as a goal was to be able to late bind connections to services. That is, apps bind on serviceIDs and the particular service instance’s network address is determined by resolving through successive service controllers. Although we could have modified each transport protocol separately, providing a standard solution in the SAL made more sense. Further, since today’s transport protocols need to read network-layer addresses for demultiplexing purposes, changes are necessary to fully support migration and mobility. Another limitation of today’s transport layer is the limited signaling they allow. TCP extensions (e.g., MPTCP and TCP Migrate) typically implement their signaling protocols in TCP options for compatibility reasons. However, these options are protocol specific, can only be piggybacked on packets in the data stream, and cannot consume sequence space themselves. Options are also unreliable, since they can be stripped or packets resegmented by middleboxes. To side-step these difficulties, the SAL uses its own sequence numbers for control messages.

We were presented with two approaches for rewiring the stack: using UDP as a base for the SAL (as advocated in [11]), or using our own “layer-3.5” protocol headers. The former approach would make our changes more transparent to middleboxes and allow reuse of an established header format (e.g., port fields would hold flowIDs). However, this solution requires “tricks” to be able to demultiplex both legacy UDP packets and SAL packets when both look the same. Defining our own SAL headers therefore presented us with a cleaner approach. We also offer optional UDP encapsulation for traversing legacy NATs and other middleboxes, which otherwise might drop

TCP	Mean	Stdev	UDP	Tput	Pkts	Loss
Stack	Mbit/s	Mbit/s	Router	Mbit/s	Kpkt/s	Loss %
TCP/IP	934.5	2.6	IP Forwarding	957	388.4	0.79
Serval	933.8	0.03	Serval	872	142.8	0.40
Translator	932.1	1.5				

Table 4.1: TCP throughput of the native TCP/IP stack, the Serval stack, and the two stacks connected through a translator. UDP routing throughput of native IP forwarding and the Serval stack.

or mishandle unfamiliar protocols. Recording addresses in a SAL extension headers also helps comply with ingress filtering (§4.3).

Since Serval uses ECCP in the SAL, the transport layer does not perform connection establishment, management, and demultiplexing. Despite this seemingly radical change, we could adapt the Linux TCP code with few changes. Serval only uses the TCP functionality that corresponds to the **ESTABLISHED** state, which fortunately is mostly independent from the connection handling. In the Serval stack, packets in an established data stream are simply passed up from the SAL to a largely unmodified transport layer. If anything, transport protocols are *less* complex in Serval, by having shared connection logic in the SAL. Our stack coexists with the standard TCP/IP stack, which can be accessed simultaneously via `PF_INET` sockets.

4.1.2 Performance Microbenchmarks

The first part of Table 4.1 compares the TCP performance of our Serval prototype, utilizing ECCP, to the regular Linux TCP/IP stack. The numbers reflect the average of ten 10-second TCP transfers using `iperf` between two nodes, each with two 2.4 GHz Intel E5620 quad-core CPUs and GigE interfaces, running Ubuntu 11.04. Serval TCP is very close to regular TCP performance and the difference is likely explained by our implementation’s lack of some optimizations. For instance, we do not support hardware checksumming and segmentation offloading due to the new SAL

headers. Furthermore, we omitted several features, such as SACK, FACK, DSACK, and timestamps, to simplify the porting of TCP, but can be included in the future. We speculate that the lack of optimizations may also explain Serval’s lower stdev, since the system does not drive the link to very high utilization, where even modest variations in cross traffic would lead to packet loss and delay. The table also includes numbers for our translator (§4.3), which allows legacy hosts to communicate with Serval hosts. The translator (in this case running on a third intermediate host) uses Linux’s `splice` system call to zero-copy data between a legacy TCP socket and a Serval TCP socket, achieving high performance. As such, the overhead of translation is minimal.

The second part of Table 4.1 depicts the relative performance of a Serval service router versus native IP forwarding. Here, two hosts run `iperf` in unconnected UDP mode, with all packets forwarded over an intermediate host through either the SAL or just plain IP. Throughput was measured using full MSS packets, while packet rate was tested with 48-byte payloads (equating to a Serval SYN header) to represent resolution throughput. Serval achieves decent throughput (91% of IP), but suffers significant degradation in its packet rate due to the overhead of its service table lookups. Our current implementation uses a bitwise trie structure for LPM. With further optimizations, like a full level-compressed trie and caching (or even TCAMs in dedicated service routers), we expect to bridge the performance gap considerably.

4.1.3 Application Portability

We have added Serval support to a range of network applications to demonstrate the ease of adoption. Modifications typically involve adding support for a new `sockaddr_sv` socket address to be passed to BSD socket calls. Most applications already have abstractions for multiple address types (e.g., IPv4/v6), which makes adding another one straightforward.

Application	Version	Codebase Size	Serval Changes
Iperf	2.0.0	5,934	240
TFTP	5.0	3,452	90
Wget	1.12	87,164	207
Elinks browser	0.11.7	115,224	234
Firefox browser	3.6.9	4,615,324	70
Mongoose webserver	2.10	8,831	425
Apache Bench / APR	1.4.2	55,609	244

Table 4.2: Applications currently ported to Serval. Codebase size and Serval changes measured in lines of code.

Table 4.2 overviews the applications we have ported and the lines of code changed. Running the stack in user-space mode necessitates renaming API functions (e.g., `bind` becomes `bind_sv`). Therefore, our modifications are larger than strictly necessary for kernel-only operation. In our experience, adding Serval support typically takes a few hours to a day, depending on application complexity.

4.2 Experimental Case Studies

While Serval provides many benefits to both backends and clients, in this section we focus on case studies pertaining to this thesis, i.e., those utilizing ECCP’s support for mobility. Other evaluation case studies involving replicated web services and scalable backend services are evaluated in the Serval paper [24].

In a cloud setting, webserver may run in virtual machines that can be migrated between hosts to distribute load. This is particularly attractive to “public” cloud providers, such as Amazon (EC2) or Rackspace (Mosso), which do not have visibility into or control over service internals. Traditionally, however, VMs can be migrated only within a layer-2 subnet, of which large datacenters have many, since network connections are bound to fixed IP addresses and migration relies on ARP tricks. Further, this (limited) mobility is only supported for the backends; clients cannot migrate between networks even when it would be useful, such as a student crossing

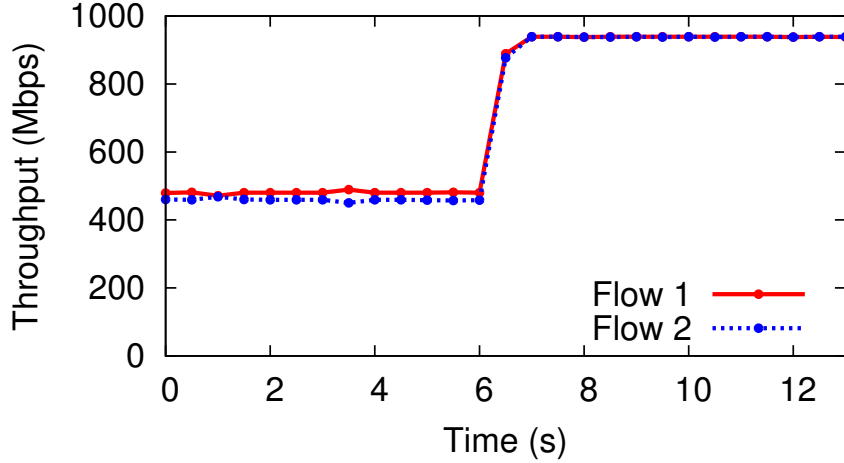


Figure 4.1: A Serval-enabled host migrates one of the flows sharing a GigE interface to a second interface, yielding higher throughput for both.

a campus with spotty WiFi while trying to stream media to their phone. To that end, we show how Serval can be used to balance network load, enable VM migration across IP subnets, and allow a phone to switch networks seamlessly.

Interface load balancing: Modern devices, including smartphones on the client side and commodity servers in datacenters, have multiple physical network interfaces. Because of ECCP, a Serval-enabled device can **accept** or **connect** using one interface, and then migrate the flow to a different interface (possibly on a different layer-3 subnet) without breaking connectivity. To demonstrate this functionality, we ran an `iperf` server on a host with two GigE interfaces. Two `iperf` clients then **connected** to the server and began transfers to measure maximum throughput, as shown in Figure 4.1. Given TCP’s congestion control, each connection achieves a throughput of approximately 500 Mbps when connected to the same server interface. Six seconds into the experiment, the server’s service controller signals the SAL to migrate one flow to its second interface. TCP’s congestion control adapts to this change in capacity, and both connections quickly rise to their full link capacity approaching 1 Gbps. Given the right conditions—i.e., plenty of battery and a need for high throughput—mobile devices would find this ability useful as well, which we explore more in §6.2.3.

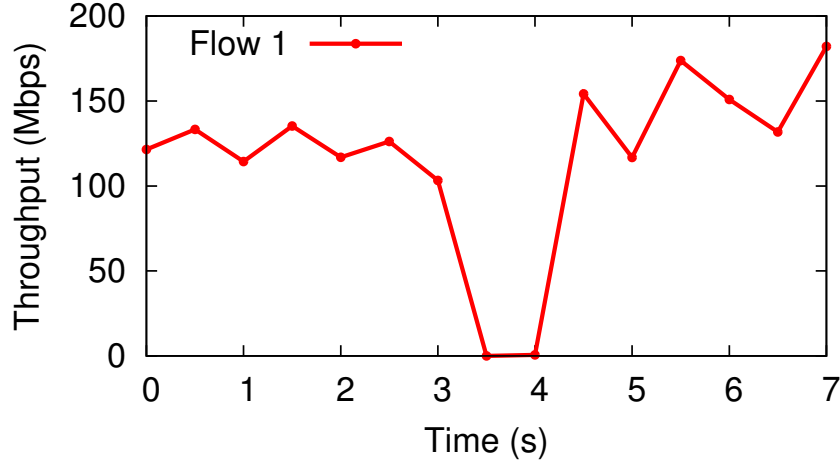


Figure 4.2: A VM migrates across subnets, causing a short interruption in the data flow.

Virtual machine migration: Cloud providers can also use Serval’s migration capabilities to migrate virtual machines across layer-3 (i.e., the network/IP layer) domains. Figure 4.2 illustrates such a live VM migration that maintains a data flow across a migration from one physical host to another, each on a different subnet. Once again we use `iperf` where a client is connected measuring the bandwidth. After the VM migration completes (just after 3s), TCP stalls for a short period, during which the VM is assigned a new address as it gets a new address from DHCP on the new network ¹. Afterwards it performs an RSYN handshake and the connection continues as before.

Mobile clients: Serval’s use of ECCP for its migration underpinnings makes it a useful fit for mobile devices. For example, seamless music streaming, with opportunistic WiFi off-loading, on a single, continuous flow is very convenient for data cap conscious users. Figure 4.3 shows a map of network usage for a mobile user who is walking across a college campus. A single music stream is migrated back and forth between cellular (gray spots) and WiFi (red spots), offloading some data usage while not losing connectivity.

¹If the VM had statically assigned IPs on both networks, this gap could be reduced and perhaps eliminated.



Figure 4.3: The thesis author uses a Serval-enabled phone to stream music while walking across campus. The phone migrates the connection between available WiFi (red) and cellular 4G (gray) networks, without loss of playback quality. The opacity of each data point is an indicator of the throughput (normalized within network type) achieved at that location.

4.3 Incremental Deployment

This section discusses how Serval can be used by unmodified clients and servers through the use of TCP-to-Serval (or Serval-to-TCP) *translators*. This allows us to support unmodified applications and/or end-hosts. For both, the application uses a standard PF_INET socket, and we map legacy IP addresses and ports to serviceIDs and flowIDs.

Supporting unmodified applications: If the end-host installs a Serval stack, translation between legacy and Serval packets can be done on-the-fly without terminating a connection: A virtual network interface can capture legacy packets to particular address blocks, then translate the legacy IP addresses and ports to Serval identifiers.

Supporting unmodified end-hosts: A TCP-to-Serval translator can translate legacy connections from unmodified end-hosts to Serval connections. To accomplish this on the client-side, the translator needs to (i) know which service a client desires to access and (ii) receive the packets of all associated flows. Several different deployment scenarios can be supported.

To deploy this translator as a client-side middlebox, one approach has the client use domain names for service names, which the translator will then transparently map to a private IP address, as a surrogate for the serviceID. In particular, to address (i), the translator inserts itself as a recursive DNS resolver in between the client and an upstream resolver (by static configuration in `/etc/resolv.conf` or by DHCP). Non-Serval-related DNS queries and replies are handled as normal. If a DNS response holds a Serval record, however, the serviceID and FORWARD rule are cached in a table alongside a new private IP address. The translator allocates this private address as a local traffic sink for (ii)—hence subsequently responding to ARP requests for it—and returns it to the client as an **A** record.

Alternatively, large service providers like Google or Yahoo!, spanning many data-centers, could deploy translators in their many Points-of-Presence (PoP). This would place service-side translators nearer to clients—similar to the practice of deploying TCP normalization and HTTP caching. The translators could identify each of the provider’s services with a unique public IP:port. The client could resolve the appropriate public IP address (and thus translator) through DNS.

As mentioned in §4.1.2, we implemented such a service-side TCP-to-Serval translator [29]. When receiving a new client connection, the translator looks up the appropriate serviceID, and initiates a new Serval connection. It then transfers data back-and-forth between each socket, much like a TCP proxy. As shown in our benchmarks, the translator has very little overhead.

A Serval-to-TCP/UDP translator for unmodified servers looks similar, where the translator converts a Serval connection into a legacy transport connection with the server’s legacy stack. A separate liveness monitor can poll the server for service (un)registration events.

In fact, both translators can be employed simultaneously, e.g., to allow smartphones to transparently migrate the connections of legacy applications between cellu-

lar and WiFi networks. On an Android device, `iptables` rules can direct the traffic to any specified TCP port to a locally-running TCP-to-Serval translator, which connects to a remote Serval-to-TCP translator,² which in turn communicates with the original, unmodified destination. This is the approach we used for Figure 4.3.

Handling legacy middleboxes: Legacy middleboxes can drop packets with headers they do not recognize, thus frustrating the deployment of Serval. To conform to middlebox processing, Serval can optionally encapsulate SAL headers in shim UDP headers, as described in §4.1. The SAL records the addresses of traversed hosts in a “source” extension of the first packet, allowing subsequent (response) packets to traverse middleboxes in the reverse order, if necessary.

²In our current implementation of such two-sided proxying, the client’s destination is inserted at the beginning of the Serval TCP stream and parsed by the remote translator.

Chapter 5

Managing Limited Resources with Tango

Mobile devices come with more limitations on resources than previous devices. Since these devices are used on the go, battery life has to support long periods of time away from charging opportunities (i.e., hours or most of a day) and having ubiquitous network connectivity necessitates wide-range technologies like cellular, which often come with a data plan that must be rationed throughout a month. Failing to do so leaves a user with a device that cannot be fully utilized, or worse, utilized at all.

The current state of affairs, as seen in Table 5.1, shows users have a variety of knobs they can tune to try to manage these resources. But this is a problem itself—users must micromanage each new app to make sure it conforms to whatever resource preferences the user has. How well each app conforms depends on the settings exposed, which can be added, changed, and removed with app updates. Further, the options usually offer limited level of control (e.g., whether to use mobile networks fully or not at all) whereas users’ preferences are likely more fluid (e.g., use mobile if not near data cap). Managing all these settings over varying epoch lengths (i.e., hours for battery, days for data caps) quickly becomes burdensome and instead users

App class	App name	Settings
Social	Facebook	Refresh rate: None / 30m / 1h / 2h / 4h Sync photos: Any network / WiFi only
	Google+	Sync photos: Any network / WiFi only Sync videos: Any network / WiFi only Sync while roaming: On / Off Sync only while charging: On / Off
Audio Streaming	Pandora	High-quality on cell: On / Off Conserve battery: On / Off
	Google Play Music	Cache during playback: Yes / No Auto cache while charging + Wifi: Yes / No Pin songs on WiFi only: Yes / No Stream on WiFi only: Yes / No Cell stream quality: Low / Normal / High
Video Streaming	Youtube	HD on cell: Yes / No Uploads: Any network / WiFi only Preload WiFi + Charging: None / Subscriptions / Watch Later / Both
	Netflix	Playback on WiFi only: Yes / No

Table 5.1: Application settings available for managing resource usage.

settle for suboptimal usage. Finally, with the introduction of ECCP/Serval or other systems that improve mobility in today’s networks [38, 27, 42, 10], a more expressive system of deciding *which* networks to use and *when* to use them is necessary.

A better management experience would be for the user to be able to express preferences they have over device-level resources (i.e., a policy of resource management), and then have the operating system manage resources according to those preferences. The preferences, or policy, could dictate the level of usage of different resources, including which networks to use, which rates to use on those networks, and/or prioritization of different types of traffic. Apps could hook into these preferences, and tune their settings to match a user’s preference. Since apps also have domain knowledge of how they are using the network, they can use that to further optimize network usage.

To that end, we introduce Tango. Tango centralizes network management in a controller process that monitors device state and executes dynamically-generated actions that incorporate both user and app-specific needs. This controller both distills ad-hoc user configuration into a single user policy and provides a means of resolving

conflicts between user and app policies. In Tango, user interests trump app interests, and a constraint mechanism is used that enforces limits (even for legacy or uncooperative apps) while *encouraging* apps to align. The constraint system *proactively* deals with policy conflicts, informing apps of their limits so they can adjust. Alternative *reactive* solutions, such as discarding or changing outputs of conflicting policies, would leave apps either uninformed if their policy was carried out or unable to know what policies are acceptable in the first place. Meanwhile, cooperative apps can leverage the constraints and local knowledge (e.g., about buffers or counters) to optimize their usage. Additionally, Tango allows apps to “hint” about their needs (e.g., priority, data rate) that become part of device state. These hints can be incorporated by the user policy, allowing apps to provide feedback while still maintaining user policy preference.

A Tango *policy* is a program that, given the device state and constraints, outputs a list of actions to take on an entity’s behalf (i.e., a user or an app). Because mobile devices’ operating conditions are quite dynamic, Tango collects information from a variety sources—including the kernel, network stack, and battery—to be used as input to policies. For example, a user may prefer using WiFi over 4G, except when WiFi provides unacceptable performance. When considering whether to use WiFi or 4G, a policy can consider the number of apps using the network, whether there is any foreground activity, and the current traffic demands on the network. This model is considerably richer and more flexible than the options currently available.

5.1 Motivation and Challenges

Today, music streaming services, like Pandora, Rdio, Spotify, Google Play Music, and iTunes Radio, are popular alternatives to preloading devices with large music libraries. In fact, a 2011 study [39] found that media streaming (including music) is

one of the major sources of cellular data usage, accounting for 5-15% of data, and is typically the largest single category. Using music streaming as a motivating example, we next highlight some areas in which current network management does not afford a good user experience. We also discuss challenges in aligning the interests of users, device vendors, and app developers.

5.1.1 Balancing Costs, Caps, and Battery

With unlimited data plans a thing of the past, the increased flexibility of streaming comes with the risk of inflating cellular data usage and power consumption. Prefetching upcoming songs to provide a seamless playback experience users expect inevitably leads to trade-offs, where the data limited and power hungry cellular link must be used.

Previously, table 5.1 showed that apps try to address managing these trade-offs by providing many knobs for the user to tune. However, this forces the user to choose reduced functionality or constant micro-management. The level of control exposed is left up to app developers, making it inconsistent even across apps of the same class (e.g., Pandora vs. Google Play Music). Further, some app settings assume a certain level of understanding of the inner workings of the app, and settings may come and go as the app is updated. Finally, a user's interests can change over the course of a billing cycle depending on their data use, or over the course of the day as their battery drains. Trying to balance all these concerns, via numerous, redundant, inconsistent settings and over an entire billing cycle, becomes an insurmountable problem for many users. With Tango, many of these settings are distilled into policies that are configurable from a single location via the user policy. Still, Tango supports app-specific settings, but may override them if they conflict with the user policy.

5.1.2 Ensuring Good User Experience

Smartphones automatically switch between networks (e.g., 4G and WiFi), but this can be disruptive to the user experience. Network switching is done with little regard to its effect on apps, which typically experience failed TCP connections when IP addresses change. For streaming music, this manifests as pauses in playback at best; other apps can see broken webpages, disconnects from a game, or other failures. While every app could deploy its own failure handling, this leads to little economy of mechanism and places an undue burden on developers. Further, efficient recovery is not always supported by remote servers (e.g., only about 39% of the top ten thousand Alexa websites supports HTTP range requests [26]). In the worst case, we found some popular streaming apps that simply fail and tell the user to try again.

This problem is exacerbated by the aggressive WiFi offloading performed by today’s smartphones, even when performance on WiFi is worse than cellular. Examples of this include public hotspots that are overloaded with too many users, networks with weak signal, or networks with a low-bandwidth backhaul link. Without a way to seamlessly switch between networks and an effective policy for when to switch, the user is left to either manually manage their connectivity, live with the poor performance, or inflate cellular usage by never using WiFi. Tango improves the user experience by allowing data streams (e.g., music playback) to continue seamlessly across network changes (e.g., by adopting Serval [24] or MPTCP [38]) and having policies dynamically pick the best network according to user interests. Tango’s monitoring of device state, including how networks are performing, can help migrate from poor networks that are not meeting user requirements (e.g., not enough bandwidth for music streaming) more precisely; users do not have to make ad-hoc decisions about whether a network feels “slow.”

5.1.3 Managing Conflicting Interests

Conflicting user and app interests are another source of tension on mobile devices. Apps typically focus on ensuring they perform well, even at the expense of wasted data usage. This is particularly a problem with streaming apps, e.g., where prefetched content may be discarded when the user skips songs or stops playing, or is unnecessarily prefetched because the user reaches WiFi before it is needed. Apps may offer options to disable functionality under certain conditions—or try to do so automatically—but these are usually coarse-grain options without the ability for finer tuning.

Interest conflicts also occur among concurrently running apps. Some mobile OSes limit the types of background apps (e.g., media, location tracking) in an attempt to minimize these conflicts. Yet such coarse-grain policy inhibits non-whitelisted apps, while still failing to provide sufficient resource isolation. For example, streaming apps will prefetch songs even if their need is not imminent, slowing down interactive foreground tasks such as web browsing. Also, because an app’s usage may be spread over many flows—e.g., one for streaming media and another for prefetching—traditional resource-management policies like per-flow fairness may be insufficient. Further, resource prioritization can be more nuanced than a mere differentiation between foreground and background apps. For instance, the data usage of a music streaming app in the background is as important as the foreground task when the amount of music buffered is low since playback could stop.

Tango supports the restriction and prioritization of resources on a per-app basis. By considering apps rather than flows, it prevents any “strength-in-numbers” attempts by apps to gain an unwarranted portion of network resources. User policy can dynamically prioritize apps on their execution status, as well as consider hints from apps about when to re-prioritize their usage (e.g., when a background music app signals the need for higher priority because its buffer has fallen below a low watermark). Additionally, Tango’s constraint system lets apps know how to not in-

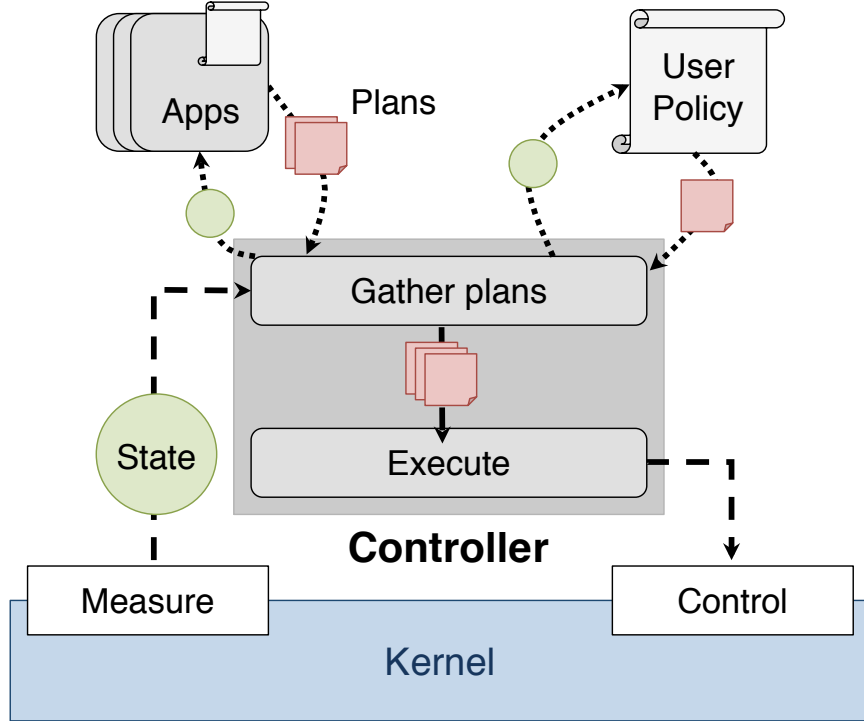


Figure 5.1: The Tango architecture. Device state is continuously collected from sources in the device kernel, packaged by the controller (center gray box), and evaluated by all registered policies. Each policy outputs a list of actions (e.g., rate limit) called plans, which are carried out by the controller using tools provided by the kernel and operating system.

terfere with user preferences, so apps can fine tune their usage without sacrificing performance unnecessarily.

5.2 Tango Design

This section details Tango’s design and how it manages network usage. Figure 5.1 shows a high-level overview of Tango’s architecture. At the heart of Tango is a controller that runs as a privileged process, used to centralize network control and to handle the dynamic nature of mobile devices. The controller is responsible for monitoring and packaging device state into a common API used by policies. A *policy* in Tango refers to a programmatic instantiation of an entity’s interests. Policies use

Pseudocode 1 Tango Control Loop

```
1: A: applications
2: for every epoch do
3:   S  $\leftarrow$  MEASUREDEVICESTATE
4:   C  $\leftarrow$  DETERMINECONSTRAINTS(S)
5:   ENFORCE(C)
6:   for a  $\in$  A do
7:     SOLICITPLAN(a, C)
8:   end for
9:   P  $\leftarrow$  GATHERPLANS(A)
10:  for p  $\in$  P do
11:    if not VALID(p, C) then
12:      p  $\leftarrow$  GETDEFAULTPLAN
13:    end if
14:    EXECUTE(p)
15:  end for
16: end for
```

device state to derive a *plan*, a list of actions to be carried out on the entity’s behalf. These plans are vetted by the controller to make sure they are valid. Valid plans do not contain incompatible actions (e.g., setting two different rate limits), do not act outside an entity’s scope (e.g., an app policy trying to migrate another app’s flow), and obey *constraints* set by a higher-priority policy (i.e., from the user). Constraints serve as a mechanism to proactively resolve interest conflicts between a user and an app (or between apps) by providing limitations on usage by an app (e.g., “rate limit of 200 kbps on 4G”). In the rest of this section, we detail the responsibilities and benefits of the controller, the role of user and app policies, and look at policy in practice. Throughout this section we include pseudocode examples to give a sense of what policies look like and how simply many of these concepts can be expressed in our framework. While we use pseudocode here, the actual policies are very similar in terms of complexity and length; the pseudocode is just more terse than the language in which are prototype policies are written (Java, see §6.1).

Network stack	
Transport	# of retransmissions, RTTs, congestion window, etc.
Network (IP)	addresses, routing rules, etc.
Physical	link type, signal quality, bit errors, etc.
Other sensors	
Battery	plugged in, charge percent, current draw, etc.
GPS	location, speed

Table 5.2: Device state sources and metrics.

5.2.1 The Controller and Policy Execution

The controller process is at the heart of Tango, designed to be a central point for network management. The controller’s control loop is presented in Pseudocode 1. This loop is run once per control epoch, which is configurable and typically on the order of seconds. Alternatively the loop could run in an event-based manner, responding to device state changes like new interfaces becoming available, particular apps opening, etc.

Monitoring device state. The control loop starts with the controller compiling a current view of the device state, which it exposes as a high-level API to policies. This provides apps with a common way to access device state, rather than leaving the implementation up to each app. The device state is composed of metrics from numerous sources including the OS, the network stack, battery, apps, and optionally other available sensors, as shown in Table 5.2. Reading some of these sources may require elevated privileges, so by centralizing this process we also remove the need for apps to request additional permissions. Further, Tango enforces information protection by only sharing relevant state with an app, e.g., the transport-layer statistics for its flows.

This device monitoring stage is important for handling the dynamic environments in which mobile devices are used, as it allows the controller to stay up-to-date with

operating conditions. While current network management mechanisms recognize the importance of certain state (e.g., WiFi signal strength), Tango increases the scope of monitored state; bandwidth, latency, app foreground status, etc., provide a more complete picture of the current environment. For example, in the case of overcrowded public hotspots, available bandwidth is a much more important metric for quality than signal strength. The information supplied by Tango allows for more versatile and useful policies.

Enforcing constraints. One task of the user policy is to specify constraints to the system. Constraints are limits on system resources that are enforced on entities such as interfaces and apps. For example, to reduce data usage, a user policy could constrain the 4G interface to 500 kbps, which the controller applies using system tools and APIs (§6.1). The controller rejects actions generated by policies attempting to violate this, e.g., an app trying to set its limit to 1Mbps. In this manner, a user’s interests are enforced even in the presence of misbehaving apps.

Tango uses this constraint mechanism to proactively resolve conflicts between competing interests. In Tango, user interests trump those of apps, which is why the user policy determines these constraints. Constraints are intended to allow app policies to cooperate with user policy, but are also always enforced by the controller, even if a malicious app policy tries to skirt them. By telling app policies of constraints upfront, apps can optimize their usage within those limits. For example, a user may rate limit a streaming app to marginally above its playback rate; the app adjusts by de-prioritizing non-critical flows (i.e., prefetching) in favor of critical ones (i.e., streaming). If a user’s constraints cause the app to perform poorly, either the user policy needs to be refined to be less restrictive, or the app’s interests are too divergent. We imagine that frequently conflicting policies can hurt an app’s online feedback, incentivizing apps to include policies that work well across operating conditions.

We preferred this proactive approach, with its straightforward controller task of approving app plans, over attempting to resolve conflicts after app policies had been evaluated. To do that, conflicting plans would need to be modified by the controller or use a feedback loop that involves reevaluating app policies. The former leaves apps optimizing for conditions that may not happen, while the latter also need a constraint mechanism to avoid the repeating conflicts in subsequent rounds. Additionally, we chose to make user policy trump app policy because the device belongs to the user and *they* will ultimately “pay” the consequences of resource mismanagement (e.g., data overage costs, the battery dying before the end of the day). From this standpoint, apps that fail to comply with user policy can be thought of as *violating correctness*. App hints and policies allow some room for compromise, however, within the confines specified by the user policy’s constraints.

Policy conflicts also occur across different apps. In such situations, constraints are useful for expressing prioritization and how usage should be shared amongst the conflicting apps. Since a constraint applies on a per-app basis, rather than per-flow, it is not possible for apps to game the system by a “strength in numbers” approach. That is, creating multiple flows to gain more bandwidth is futile. App priority can be specified by allocating larger resource shares to high-priority apps, or by limiting lower-priority apps while leaving others unrestricted.

Supporting app policies. Tango allows apps to specify their own policies. During each epoch (lines 6-8 in Pseudocode 1), the controller disperses appropriate device state and constraints to apps registered with the controller. The app policy responds with a plan, which the controller validates (lines 9-12), ensuring the app only manages its own flows or other approved resources and obeys its given constraints. The user policy provides a default plan for apps without one or an invalid one.

App policies allows apps to refine their network usage based on local information that a user policy would not know. For example, if a user policy restricts an app’s data limit, the app policy can respond by asking the controller to re-prioritize certain flows higher (e.g., those downloading a social network feed) than others (e.g., background syncs). In this way, apps are given more insight into what is happening to their network usage and given a way to respond in useful ways. Should an app not provide a policy, the typical default plan by user policies would be a bare minimum approach. That is, enforce the constraints on that app, but little else.

In addition to actions to take on its behalf, apps can provide hints to the controller about its needs. For example, when a music app’s playback buffer is low, it can send a hint that it wants higher priority for its traffic. This information is stored as part of the device state for the next control epoch, which can be used by the user policy as part of its constraint generation process. Pseudocode 2 and 3 are examples of app and user policies using app hints. The app policy sets `hintPriority` in its plan. In the next epoch, the user policy uses `AllowPriority()` to decide if the app’s request is allowable, e.g., by matching against a list of apps and their acceptable priority levels. Apps therefore can provide feedback to the user policy, which still has the final approval of what hints to use. We look more at the usefulness of app hints when it comes to app conflicts in §6.2.3.

5.2.2 A Programmatic Approach to Policy

User policies reflect the overall desires of the device owner, which may reflect high-level interests such as “preserve battery,” “minimize cellular usage,” or “ensure high throughput for video.” Management of a device’s interfaces, and how usage is shared or prioritized across different (classes of) apps, are expressed by the user policy. User policies can naturally be written with classes of apps (e.g., music streaming apps) in mind; the class of each app can be prepopulated by the policy writer, configured

Pseudocode 2 App policy with hints

```
1: PS: music player state
2: urgent: urgent need for data
3: function EVALUATE(S, C)
4:   P: plan
5:   // Rest of policy elided for space.
6:   P.hintPriority  $\leftarrow$  NORMAL
7:   if PS.getBufferTime() >30 then
8:     urgent  $\leftarrow$  false
9:   else if PS.getBufferTime() <20 ——— urgent then
10:     urgent  $\leftarrow$  true
11:     P.hintPriority  $\leftarrow$  HIGH
12:   end if
13:   return P
14: end function
```

Pseudocode 3 User policy with app hints

```
1: function DETERMINECONSTRAINTS(S)
2:   C: constraints
3:   for A in S.apps() do
4:     if ALLOWPRIORITY(A, A.hintPriority) then
5:       C  $\leftarrow$  NEWAPPCONSTRAINT(A, A.hintPriority)
6:     else
7:       C  $\leftarrow$  NEWAPPCONSTRAINT(A, NORMAL)
8:     end if
9:   end for
10:  return C
11: end function
```

by the user in settings, or suggested by the app developer. This allows common constraints and goals (e.g., reduce data usage) to be set once for several apps, and reduces the complexity of policies by not focusing on individual apps. There is only one user policy running at a time, and it cannot be changed by third-party apps.

We imagine there are a few ways for users to select a policy. Device manufacturers, or even tech-savvy users themselves, could write user policies, which can then be configured and activated in the device settings. Policies could be shared by uploading them to a “policy store,” where other users could download and review them. When a user loads a policy, it can expose configuration options that users can tune to fit their needs (e.g., monthly data budget, desired music quality, etc.).

App policies, on the other hand, allow the system to reflect the needs to currently executing applications. They may use information only visible or semantically mean-

ingful to the app, e.g., a streaming app’s policy examines the playback buffer when expressing “do not buffer when above X seconds.” By soliciting plans from apps, the controller can account for such app-specific information in its control loop, provided it does not violate the constraints set by the user policy. Apps can only specify policy which affects their usage; they can not manipulate usage of other apps or set the user policy. In our experience (§6.2), adding policy to apps was straightforward and a matter of exposing the pertinent information (e.g., buffer levels) to the app policy via shared state. Also, since the actions are handled by the controller, there were less permissions and code needed for the app itself.

Implementing and expressing policies. Policies are programs that implement a simple interface. This interface consists of an `evaluate()` function that constructs a plan (a list of actions) given (1) the current device state, e.g., network metrics, available interfaces, battery life, (2) constraints such as bandwidth limits, and (3) a list of controllables. Additionally, the user policy’s `determineConstraints()` function returns interface and app constraints based on the current device state.

A plan is a list of actions that a policy would like executed. Tango currently exposes two types of controllable entities: *interfaces* and *flows*. Actions on interfaces include turning interfaces on and off, selecting access points (APs), and setting queue and rate limits. Interface actions are only available to the user policy, as they affect all apps on the device. Actions on (sub)flows include adding/removing flows and migrating them across interfaces. These actions are available to both user and app policies, though app policies are restricted to acting only on their own flows. Table 5.3 summarizes these actions.

Tango’s programmatic approach provides sufficient flexibility. It allows for simple, rule-based approaches (like current techniques), as well as more complicated approaches that use past behavior to predict future usage. An example policy snip-

Action	Interface	Flow	Description
ENABLE	✓	✓	Enable interface/subflow
RATELIMIT	✓	✓	Limit bandwidth
LOG	✓	✓	Write information to file
MANAGE	✓		Change access point, queue size, etc
MIGRATE		✓	Move flow to different interfaces

Table 5.3: Actions on interfaces and flows. App policy can only perform flow actions and only on their flows.

Pseudocode 4 Avoid poor WiFi

```

1: sigs: list of WiFi signals
2: slowNets: map networks to time added
3: function EVALUATE(S, C)
4:   P: plan
5:   wifi ← GETWIFIINTERFACE(S)
6:   cell ← GETCELLINTERFACE(S)
7:   if wifi.isAssociated() then
8:     sigs.push(wifi.signal())
9:     if BADSIGNAL() then
10:      P.add(MANAGE, DISCONNECT, wifi)
11:      P.add(MANAGE, CONNECT, cell)
12:     else if wifi.speed() < 100000 then
13:      slowNets.put(wifi.network(), S.now())
14:      P.add(MANAGE, DISCONNECT, wifi)
15:      P.add(MANAGE, CONNECT, cell)
16:     end if
17:   end if
18:   // Other cases elided for space.
19:   return P
20: end function

```

pet for WiFi-connected devices is given by Pseudocode 4. Two conditions cause the policy to return a plan that fails over to 4G: (1) if `BadSignal()` return true or (2) if the measured speed of WiFi is below 100 kbps. There are many possible implementations for `BadSignal()`, including tracking a list of past signal readings to monitor trends instead of instantaneous readings (see §6.2.2). Condition (2) helps address several scenarios mentioned earlier, such as overcrowded hotspots. This is just one sample implementation; others could integrate information about the mix of apps using the network, location, battery life, and more.

5.2.3 Discussion: Policy in Practice

We now revisit the discussion of use cases from §5.1, and expand on how policy improves usage in practice.

From incidental to intentional device behavior. Today’s mobile device network management, with settings spread across apps, has at best an *incidental* effect on resource usage. Users are not certain whether their combination of settings will translate into what they want. In contrast, Tango allows users to load and run policies that have *intentional* effects on the way a device behaves. Unlike today’s myriad settings, policies express what the user wants, not how it is achieved. Tango allows control of device behavior from a single location, and settings are structured into global and class-specific ones. Global settings apply to all classes, while class settings could generate constraints for all apps of that class, such as a slider for streaming quality on cellular for all media apps. Apps may still have settings to further tune their usage, but they are subject to global constraints ensuring they are aligned with the user policy.

Improved user experience. Mobile OSes have many sensors and control surfaces that govern device behavior, but no effective way to translate that flexibility into an improved user experience. For instance, OSes that support seamless flow migration (e.g., iOS7 with MPTCP [34]) have the ability to switch networks without interrupting individual flows. However, as we have pointed out, many available networks are overloaded or experience weak signals. Unless flow migration is governed by an effective policy, this feature may have limited practical effect on user experience. Programmatic policy allows for solutions like building profiles of networks and usage over extended periods, to later inform a decision on whether to use a particular network.

Pseudocode 5 Prioritize foreground app

```
1: function DETERMINECONSTRAINTS(S)
2:   C: constraints
3:   for A in S.apps() do
4:     if A.isForeground() then
5:       C  $\leftarrow$  NEWAPPCONSTRAINT(A, HIGH)
6:     end if
7:   end for
8:   return C
9: end function
```

Living within one's means. Tango's constraint mechanism is useful for reigning in cellular usage. For data-heavy apps such as media streaming, the user policy could restrict those (classes of) apps, either with a static rate limit or by allotting them an amount of usage over a time interval. A static rate limit is useful for curtailing usage in the case of apps that do not have an app policy, but can cause poor performance if cellular service disappears (depleting the buffer) or if the rate is set too low. Assigning an allotment is similar to a rate limit, but provides apps with greater flexibility to optimize their usage. For example, an app with a sufficient media buffer can save its allotment until hitting a low watermark or for unexpected future events that necessitate a fast response, such as the user skipping the current song.

Prioritization and fair network usage. Tango makes supporting priority and fairness in network usage straightforward with constraints and the controller's state monitoring. A common example of network usage to prioritize would be that of the foreground app (changes of which are learned quickly by the controller). Achieving this prioritization in Tango can be done by implementing the user policy's `DetermineConstraints()` per Pseudocode 5. It also takes care of managing the kernel traffic queues for the user and ensures it does not cause unintended side effects with other settings. Since constraints apply on a per-app basis, Tango can effectively reign in greedy or buggy apps that would otherwise drain resources, e.g., by having many open TCP connections.

5.3 Tango Related Work

The advent of mobile computing has led to the development of several approaches to incorporating “context” into programming for richer application models. JCAF [4] is a context framework for Java applications consisting of “entities” that both make up context and respond to changes in other entities. It is not tailored for mobile and does not focus on resource management, but rather getting entities to respond in the presence of other entities. Tango instead focuses on the mobile platform and managing resources which can be quite scarce. JCAF’s somewhat open-ended nature would be make it difficult to handle things that Tango does, like the user and app policy split and our constraint model. CASS [9] is another context framework that uses nearby sensors and a remote server to create a context view to supply to apps. It abstracts away details to simplify policy writing (e.g., converting a temperature reading into a state list of “cold,” “normal,” and “hot”). Tango is instead completely local to the device, both in terms of sensing data and compiling the resultant state. Tango also leaves the level of abstraction up to policy writers by providing mostly raw metrics. CARISMA [7] is a context framework that attempts to resolve conflicts not only on-device but also amongst multiple devices using the same app. It uses utility functions and a sealed bid auction as its conflict resolution mechanism. Tango does not attempt to coordinate policies amongst multiple devices, but instead focuses on dealing with conflicts between policies on the same device. Further, our constraint mechanism helps deal with these conflicts proactively, rather than use utility functions, which are hard to define for the potentially large policy space.

Several earlier projects focus on selecting between multiple wireless networks. Ormond et al. [25] uses a utility-based approach to minimize costs when uploading files by choosing between multiple networks with varying costs and bandwidth, subject to time constraints. Wilson et al. [37] uses a fuzzy-logic inference engine that takes input from both user and apps and, based on pre-defined QoS metrics and rules,

decides on the preferred network. Ylitalo et al. [41] presents an interface selection framework where flows can be moved between several networks. It requires some changes to the socket API and uses a rule-based approach for selection. Tango’s flexible programmatic model supports these rule- and utility-based approaches, as well as considerably richer policies. Further, Tango’s app policy and hints allow for broader app input, yet still avoids OS or socket API modifications. Finally, beyond network choice, Tango addresses deeper control of the network by exposing management control of traffic queues.

Other prior work focuses specifically on reducing cellular usage through WiFi offloading. One body of research has tried to generalize application-specific prefetching strategies by providing middleware that *batches* data for download during periods of WiFi connectivity. Lee et al. [21] describes a simulated batching strategy that delays transfers in anticipation of future WiFi connectivity. Wiffler [3] employs another batching strategy that adds prediction of WiFi throughput to determine whether transfers would complete within a WiFi connectivity window. This requires prior knowledge of data sizes and accurate WiFi prediction. IMP [13] also performs batching on WiFi, but may also (pre)fetch on cellular if allowed by budget constraints that take into account battery and data usage. BreadCrumbs [22] tracks user mobility and network conditions to forecast network connectivity, and the authors discuss its use to inform prefetching and batching strategies. SALSA [30] employs similar forecasting, using an energy-delay trade-off algorithm to select the energy-minimizing link for a data transfer. These techniques are complementary to Tango’s general framework, and similar batching strategies may be adopted by specific delay-tolerant apps running on Tango to optimize their resource usage.

In contrast to this prior work, Tango can continue data transfers despite changing connectivity, relying on ECCP [2] for migrating TCP connections. Although no batching is done, the bulk of data transfers may be moved to WiFi by rate limiting

cellular links, in anticipation of future WiFi connectivity. This ensures transparent support for interactive or latency-sensitive applications (e.g., live video streaming), even if initiated while on cellular. Prefetching and excessive buffering on cellular is also avoided, which could otherwise deplete a user’s data cap or battery resources. Even so, the prior work on forecasting and link estimation could help inform Tango policies for more accurate rate limiting and migration decisions.

Recent work [26, 40, 2] has explored seamless use of heterogeneous networks using migration techniques, based on MPTCP [38] and OpenVSwitch. Tango could adopt those or alternative migration techniques, including Mobile IP [27], HIP [23], LISP [10], or TCP Migrate [32]. Unlike such work, we use flow migration as just one of many techniques that, in combination, enable interesting control plane and policy control to better utilize available networks, while simultaneously accounting for device and user needs.

Chapter 6

Tango Evaluation

In order to get a deeper understanding of the factors to consider when creating policy for devices, we built a prototype of Tango for Android and explored creating policy to deal with common resource management problems for smartphones. First, we explore how Tango can be used to express policy to improve network choice and how the interplay between user and app policies can combine to save a user precious cellular data. We use our canonical example of music streaming for a user who wants to minimize cellular data usage using WiFi offloading without micromanaging their device. Following that, we look at several examples of policy where multiple apps are competing for resources and how to achieve different goals (e.g., app equally sharing bandwidth, app prioritization) in a dynamic way using Tango.

6.1 Tango Prototype

Our Tango prototype for Android is written in Java and consists of three parts: (i) the main library with code for generating device state and APIs for policies, constraints, and actions; (ii) a client library that provides app policy support; and (iii) a controller with most of the previously described functionality.

Our prototype includes support for flow migration on Android via ECCP [2], as implemented by Serval [24] (described in Chapter 3). Serval has support for getting flow-level metrics from the transport layer, such as RTT and congestion window. While we chose this particular implementation of flow migration to use with our prototype, Tango is not dependent on it. Other methods for gathering flow-level metrics [36] and flow migration [32, 38, 40] would work as well.

Our prototype leverages many resource management mechanisms already available in Linux and Android, and thus did not require any OS modifications. The Linux traffic control tool, `tc`, provides rate limit actions and constraints; we use the hierarchical token bucket (HTB) queue extensively for enforcing constraints. Each interface starts with an overall bucket which can be rate limited via constraints or actions. Buckets are attached to interfaces for each app that needs per-app rate limiting or prioritization. Then, we use filtering rules to put all of an app’s flows on an interface into the appropriate bucket. Additional buckets can be attached to these queues to do finer-grain QoS, e.g., at the flow level. Other functionality for managing networks are done with standard Linux tools (e.g., `ip`, `iptables`, etc), or via APIs in the Android SDK (e.g., `WifiManager`).

Along with Tango, we have implemented several user policies and a few test apps that use our client library. Our Tango policies are written as Java classes that easily interface with our prototype and, in the case of app policies, the apps they belong to. One particular app we have implemented is a music streaming app that we use extensively in our evaluation. The app downloads MP3 files over HTTP using a native Serval socket,¹ supporting near instantaneous playback of partially downloaded files. Playback starts with 30 seconds of content buffered and pauses if the buffer runs too low; play resumes when the buffer again reaches 30 seconds. This functionality is

¹Serval has a proxying solution that allow for unmodified apps to be included in Tango’s planning.

based on the behavior we observed in other popular music streaming apps, such as Pandora and Google Play Music.

6.2 Case Study Evaluation

In this section, we perform two case studies that aim to address the following questions: First, how can Tango help improve the experience of using a single phone app? With music streaming as our example, we perform comparative studies of both user- and app-level policies in the face of changing environment conditions. Second, how can Tango help provide a good user experience across concurrently-running apps? In particular, we explore how policies enable us to provide dynamic fairness and/or QoS based on changing device conditions.

6.2.1 Experimental Setup

We use a Galaxy Nexus (GSM) phone running Android 4.3 for our case studies, with T-Mobile as our 4G data provider. Tango, Serval [24], and our music streaming app are installed on the phone.

To evaluate music streaming, we wanted to use the scenario of a student walking across a college campus, attempting to use WiFi to save data. However, as is typical with a wireless environment, our field measurements observed highly variable coverage and quality, even on back-to-back walks. Thus, to make meaningful comparisons across policies, we set up an emulation environment created from traces of our walks and replayed on our phone. This allowed for repeatable experiments of different policies for the same walk. The traces we use are from walks where Android chose WiFi over cellular between 75-95% of the time; however, our results show only about 20% of that time is the link able to carry data. The trace covers roughly a mile across campus that took 12 minutes to walk. We connected to the campus WiFi,

which uses the same SSID across many access points. Overall, WiFi coverage was mostly continuous in the middle of the walk and more spotty towards the ends. The traces were collected in the afternoon during the school year, so congestion levels were typical.

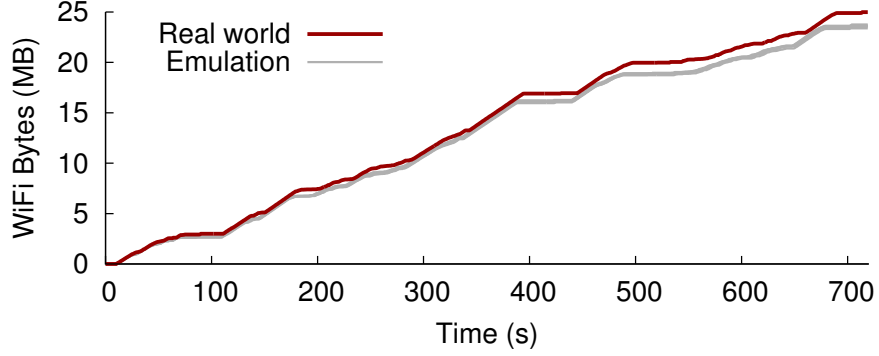
Emulation environment. We implemented the emulation using the standard Linux `tc` tool to recreate WiFi network conditions vis-a-vis packet loss and delay. The emulation leaves the portions of cellular connectivity in our traces unregulated, as we generally had no problems with cellular coverage.

To capture the variable WiFi network conditions during real walks, we used a ping-like application sending packets at a constant bit rate (CBR) to a server (one packet every 20 ms) with packet sizes to emulate TCP (1472-byte echo packets with 64-byte replies). We measured the received signal strength indicator (RSSI), upstream and downstream loss rates, and delay for every second of the walk. During emulation, the loss rates and delays were parameters for `tc`, while the RSSI readings replaced the readings from the actual WiFi driver.

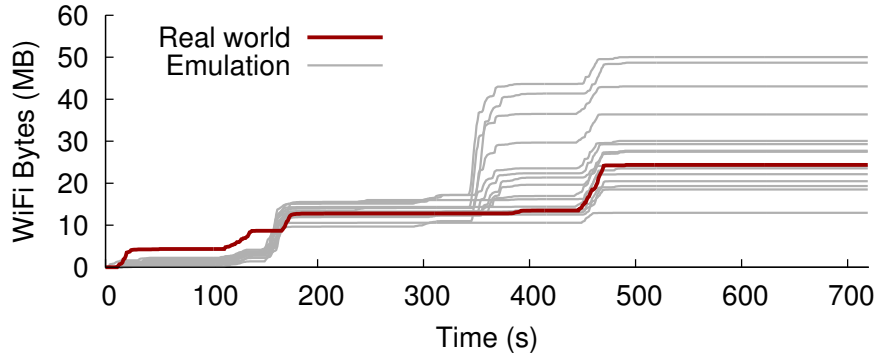
We validated the accuracy of our emulation with two types of traffic: CBR and TCP. We used the CBR traffic to assess the connectivity and drop rates and the results are shown in Figure 6.1a. Since `tc` drops packets probabilistically, we ran five emulations. There was low variance in the cumulative bytes downloaded on WiFi during these emulations (one gray line per trial). The 6% difference between the real-world and the emulations are from delays going from cellular to WiFi, i.e., DHCP and cellular teardown.² Most notably, the “shape” of the bandwidth usage is consistent across trials.

To assess the emulation’s accuracy involving TCP’s congestion and flow control, we used our music streaming app. The results are shown in Figure 6.1b, encompassing 15 emulated trials. Due to TCP’s congestion control and reliable transfer mechanism,

²The emulator reacts to logged network switching events, which in the real world are initiated 1-2 seconds prior to being logged.



(a) CBR traffic



(b) TCP traffic

Figure 6.1: Emulations of a campus walk. Both CBR traffic (a) and TCP traffic (b) emulations reproduce the real world patterns with high fidelity.

there is much greater variance in the results. Yet, the emulated trials still capture the overall “shape” of the connectivity well, i.e., they mostly share the same increases (good WiFi) and flat areas (bad WiFi). One notable difference between trials is around 350s where roughly half the trials download a significant amount of data, while the others (and real world) do not. The real world trace shows good signal quality with low drop rates and small delays in that period, but long TCP timeouts started just before good connectivity cause it to be missed during some emulations. Since a lengthy timeout starts just before good connectivity, the connectivity is wasted as no data will transmit until the timer expires. We illustrate this phenomenon in Figure 6.2. Apart from some trials having “luck” in how their TCP timers fire, all trials otherwise experience similar behavior.

(a) Retransmission timer fires *before* good connectivity (b) Retransmission timer fires *after* good connectivity

Figure 6.2: Interactions between TCP timers and changing connectivity. In (a), a lost TCP ACK packet (just before 350s) causes a 80s TCP timeout, leading to a missed opportunity to use good connectivity. In (b) some packets get through prior to 350s, causing a shorter timeout that allows use of the connectivity.

RSSI	Number of Intervals	Time %	Upstream Drop %	Downstream Drop %	Good %
(-90, -85]	54	3.88	63.56	56.97	11.11
(-85, -80]	283	20.36	55.85	55.57	9.19
(-80, -75]	367	26.40	45.00	45.67	15.80
(-75, -70]	267	19.21	33.05	31.90	34.08
(-70, -65]	180	12.95	21.06	22.68	47.22
(-65, -60]	155	11.15	10.91	10.50	75.48
(-60, -55]	60	4.32	2.54	1.12	95.00
>-55	24	1.73	2.29	0.83	100.00

Table 6.1: WiFi connectivity quality statistics across many traces of the same path. “Good” signifies both upstream and downstream had drop rates $\leq 10\%$.

Network switching. The periods of poor WiFi connectivity that cause long TCP timeouts highlight a problem with Android’s default network switching: it prioritizes WiFi too much. To this end, we developed a network switching scheme to reduce the duration on unusable WiFi. By default, Android uses WiFi whenever the RSSI is greater than or equal to -100. Rather than using an instantaneous measure, our Tango user policy tracks the last 10 seconds worth of RSSI values (one per second) and uses two heuristics to determine if the signal is degrading sufficiently to switch: (i) all 10 RSSI values have been below -75, and (ii) whether the last five were all below -80. We chose these heuristics after analyzing multiple traces at different signal levels (see Table 6.1) and testing them around campus. The measurements in Table 6.1 also

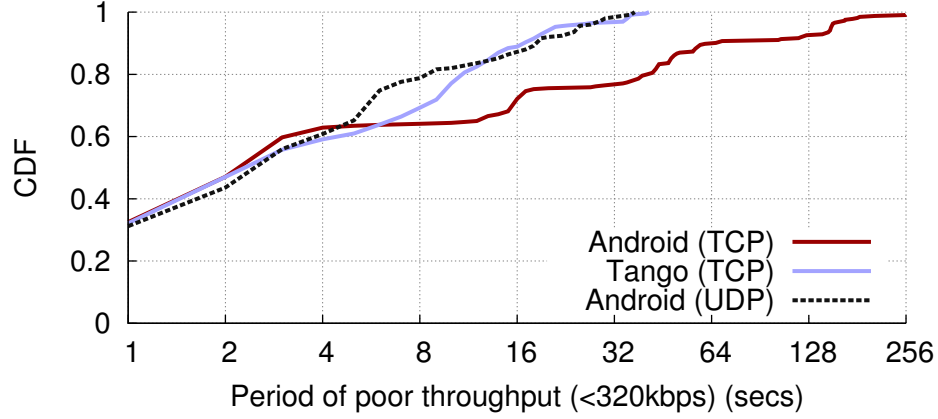


Figure 6.3: Distribution of durations of low TCP throughput, Android versus Tango switching. CBR UDP traffic serves as a baseline showing connectivity.

show that default Android’s threshold of -100 is too aggressive; more often than not, a signal strength that low is mostly unusable. While heuristics help determine when to move off WiFi, the move from cellular to WiFi relies on instantaneous readings based on regular WiFi scans. For this, we chose an instantaneous RSSI of -70 or greater as our threshold; -70 nearly doubles the amount of usable WiFi (30.15% vs 17.20%) and has an acceptable drop rate.

Figure 6.3 shows a CDF of the duration of “poor connectivity” zones on WiFi using the Android connectivity manager, our Tango switching based on the above heuristics, and the CBR baseline. “Poor connectivity” is any WiFi interval where the download rate was less than the playback rate. The CBR traffic should represent the ideal distribution since it is not subject to TCP’s timeout effects. Android’s switching has a very long tail, showing the extremely long periods of no data transfer caused by TCP timeouts. The bumps in the distribution appear to roughly correspond with when TCP timeout retries would re-establish transfer. Conversely, Tango only briefly diverges from CBR between 5 and 10s, most likely due to its heuristic to switch off WiFi after 10 seconds of poor RSSI. Tango’s pre-emptive switching prevents long TCP timeouts by moving the flow to cellular where it can continue transferring or at

	Tango switching	Constraints	App policy
Unl	✓		
Rate	✓	✓	
App	✓	✓	✓

Table 6.2: Evaluated policies.

least respond to probes. Aside for §6.2.2, our subsequent evaluation will use Tango’s switching scheme.

Configurations and policies. In the case studies, we use our emulator with the music streaming app playing at 320 kbps. The trace we use typifies a walk through our campus in terms of its connectivity and WiFi coverage.

To evaluate Tango’s use of multi-level policy, we compare both user and app policies against a baseline that allows unlimited data usage by apps, highlighting the effect of different levels of constraints and policies. The policy configurations evaluated are summarized in Table 6.2, where **Unl** allows unlimited rates (no constraints), **Rate** applies a rate-limiting constraint of 640 kbps (double playback rate), and **App** includes both user- and app-level policy. With **App**, the user policy allows the application to use some allotment of data over a given time frame. This constraint allows for bursts of traffic (for application buffering), rather than a constant limit. The app performs flow control using high and low watermarks; when the buffer goes below the low watermark, the app downloads until above the high watermark.

6.2.2 Case Study 1: Music Streaming

As discussed in §5.1, music streaming on mobile devices needs to balance data caps, costs, and battery life against the ability to provide a seamless and high-quality listening experience. WiFi offloading is a natural way to reduce cellular usage and avoid hitting data caps. Yet streaming has time requirements that do not always allow network usage to be deferred (i.e., the user wants to listen now, not wait until

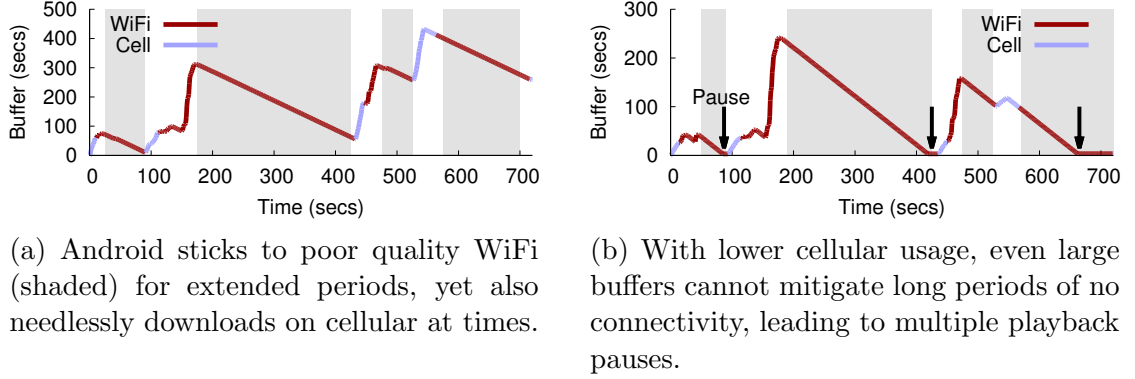


Figure 6.4: Effect of aggressive WiFi offloading on playback buffers. Android’s tendency to persist on WiFi, despite no TCP progress, leaves little room for policy to play a role in improving the application experience.

a WiFi hotspot). Thus, users are often confronted with an unfortunate trade-off between user experience and economics.

In contrast, with Tango’s seamless migration of flows, phones can switch networks when appropriate, as well as defer downloading some content until when the conditions are right on WiFi. This also works with apps that do not have their own failure and recovery mechanisms. Even with failure-handling apps, the ability to put constraints on cellular usage helps reduce costs for users, and network load for wireless providers.

Effect of Aggressive WiFi Offloading

Although WiFi offloading is desirable, it does little good unless governed by a policy that determines a good time to switch interfaces. To illustrate the (negative) effect of aggressive WiFi offloading, we first show results with Android’s default network switching, but with added flow-migration functionality. With this setup, flows are seamlessly migrated to the active interface.

Figure 6.4a shows the app’s buffer size (in seconds) during the emulated walk. While the app is able to play music without pauses, it does so with excessive cell usage, e.g., at time 527s. The buffers are well filled at this point, but because there

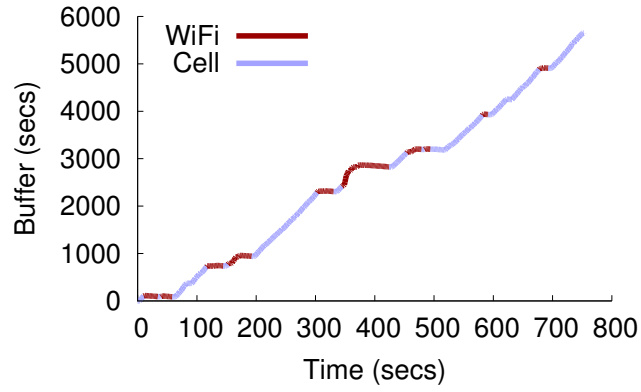
are no limits in place the app downloads indiscriminately. On this particular walk, we calculated 12MB of excess data, which can add up quickly (e.g., up to 500MB monthly if part of a twice daily walk to the office). Moreover, there are long periods of poor WiFi with no TCP progress (shaded in the graph). This is worrisome for two reasons. First, apps that cannot buffer as aggressively, such as live streaming, would likely fail many times on this walk. Second, when there is clean signal in the emulation, the client and server are on the same network, which allows more buffering than in real life over more congested wide-area links.³ Thus, with Android’s default behavior, network usage is disproportional to need on cellular, and WiFi is used inefficiently.

Unfortunately, clinging to WiFi leaves little room for reducing the cellular excess as the margin of error is small. To illustrate this, we applied a rate limit to cellular that should allow continuous playback of 640 kbps (2X playback rate). Cellular usage is reduced by almost 15 MB—a median of 19.8 MB down to 5.0 MB—but introduces playback pauses, as seen in Figure 6.4b. Because of these problems, for the rest of our experiments, we use our heuristics-based switching scheme to significantly reduce the times spent on poor-quality WiFi.

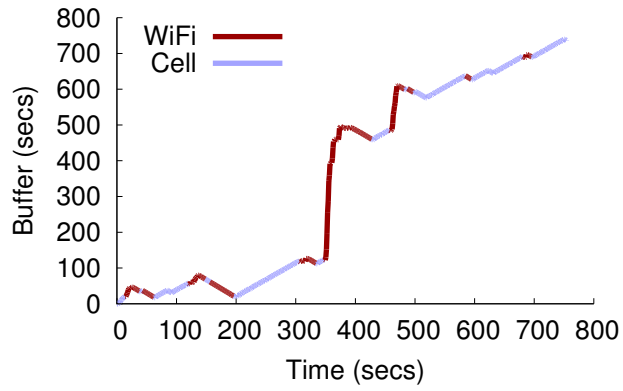
Finding a Good Policy for Music Streaming

A good policy for music streaming identifies the right “knobs” to turn, and to what extent they should be turned, in order to accommodate three goals: (i) avoid any pauses in playback, (ii) avoid needlessly using cellular bandwidth and instead use buffered content whenever possible, and (iii) avoid a significant reduction in battery life. Our study shows that the key to achieving these goals is making use of Tango’s support for multi-level policy, i.e., using input from both user and app. Further, constraints are particularly useful for reducing cellular usage by protecting against “overeager” apps.

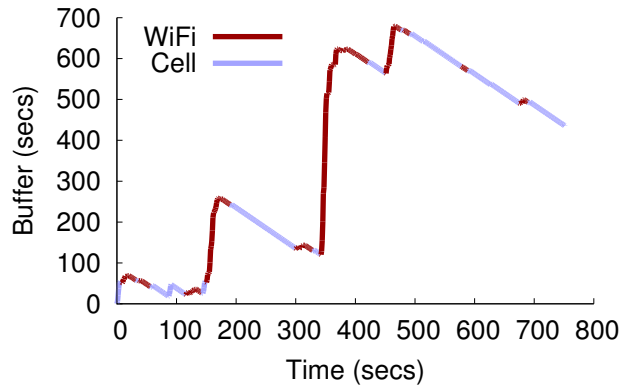
³We had the server and client on the same network for a more controlled experience across emulations.



(a) **Un1**: Buffer increases at full rate, irrespective of connectivity.



(b) **Rate**: Buffer increases at full rate on WiFi, slower rate on cellular.



(c) **App**: Buffer increases only when necessary according to app policy.

Figure 6.5: Buffer usage of different Tango policies. All avoid any pauses during playback.

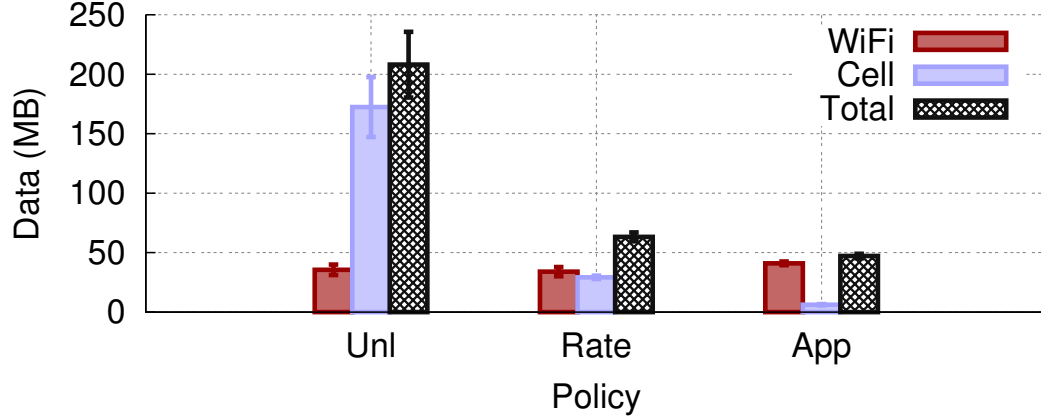


Figure 6.6: Network usage showing how Tango policies (**Rate** and **App**) can drastically reduce cellular usage compared to unlimited usage (**Unl**).

Data reductions on cellular. When evaluating our first two goals, **Unl** serves as our baseline, while **Rate** and **App** introduce user policy and multi-level policy, respectively. We can compare the buffer graphs for these configurations in Figure 6.5 with those for plain Android (Figure 6.4). These configurations all avoid any pauses, and their buffer decreases on WiFi (red) are less frequent and shorter, meaning that with this switching method data is buffering when plain Android is blocked on non-working WiFi. However, as Figure 6.6 shows, a consequence of moving off WiFi is more cellular usage—up to 6-7x compared to plain Android for this trace. This behavior, i.e., prefetching indiscriminately, is not unique to our app. We based our app’s behavior on popular music streaming apps (e.g., Google Play Music), which often buffered several songs ahead even on cellular.

Better connectivity is in general a good thing, but it is contrary to our goal of reducing cellular usage. Applying **Rate** and **App** policies drastically reduces cellular usage while maintaining a pause-free playback. However, only **App** has the right combination of user-policy constraints and app-policy knowledge to reduce cellular usage to 30% of that of plain Android (i.e., from 19.8MB down to 6.1MB), which already has “artificially” low cellular usage due to clinging to WiFi.

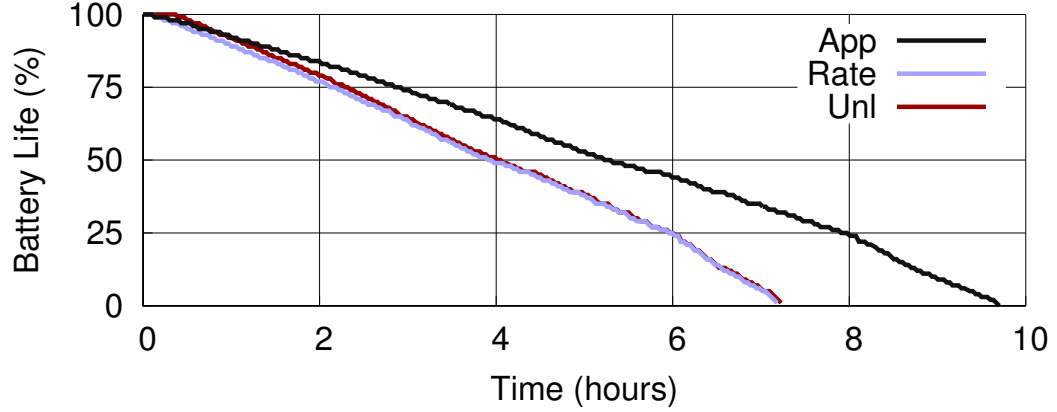


Figure 6.7: **Unl** and **Rate** drain battery faster due to keeping the cell link active, while **App** is able to reduce the drain.

Battery usage. In light of our third goal for Tango policies, we sought to understand how cellular-reducing policies affect the smartphone’s battery life. To evaluate this, we ran the music streaming app while our emulation trace looped until the battery ran out, recording the battery percentage at every second, with the results shown in figure 6.7. Also, table 6.3 breaks down the rates of decline and total life of each policy. With 25% of battery left, the drain rate appears to speed up, possibly due to OS or the firmware attempting to avoid a complete drainage, so we segment the decline rates for the first 75% and the last 25%. We see that **Unl** and **Rate** experience similar battery life—about 7.2 hours—losing $\sim 13\%$ an hour for the first 75% and $\sim 19\%$ for the remainder. This suggests that battery life is dependent on the amount of time the cellular network is active, regardless of transmission rate.⁴ On the other hand, **App** provides over 2 hours additional battery life, for a total of 9.7 hours. **App** lets the cellular radio transition to a low power idle state during times when the buffer is sufficiently full, saving power over **Unl** and **Rate**.

⁴We believe the higher drain for **Rate** is mostly noise due to external factors, i.e., the load on the cell network.

	First 75% batt. drain (% / hr)	Last 25% batt. drain (% / hr)	Battery Life (hrs)
Unl	-12.6	-19.2	7.25
Rate	-12.7	-19.5	7.21
App	-9.5	-13.9	9.70

Table 6.3: **Unl** and **Rate** keep cellular active and drain battery faster, while **App** is able to reduce the drain.

6.2.3 Case Study 2: Policy Across Apps

We now consider how Tango can provide fairness and prioritization *across* apps competing for resources.

App-level Fairness

On mobile devices, apps that open up many flows can gain a higher share of bandwidth, since TCP only provides fairness on a flow level. For example, an app downloading several songs concurrently for future listening would drown out a single-flow video stream. Since users typically think in terms of apps rather than flows, fairness at the app level can be a more natural fit for expressing a user’s needs. We can achieve this in Tango with a user policy that gives each (active) app an equal share of available bandwidth by setting a constraint on each app to have $1/N$ of the bandwidth.

We implemented this policy in a scenario with multiple networks available simultaneously—giving apps a choice of which to use—but enforcing equal sharing of each link between apps. In our scenario, the user policy rate limits the 4G network to 640 kbps after 30 seconds to discourage its use, with the constraint removed after 30 seconds of being idle. We have two apps: a multi-flow app (MFA) with five flows that always uses the 2 Mbps WiFi link, and a single-flow app (SFA) with an app policy to migrate to the network where it gets the best performance. When simultaneous

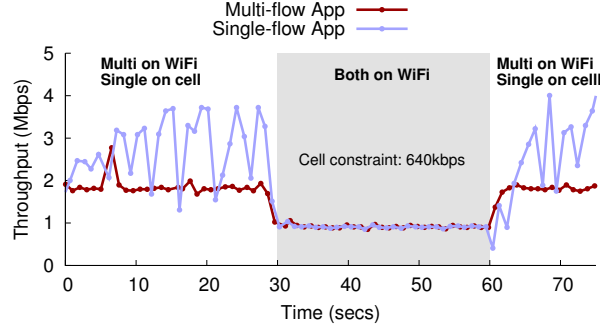


Figure 6.8: App-level fairness at link level, while app policy optimizing performance given constraints.

interface usage is acceptable, Tango helps reduce costs and/or optimizes performance by scheduling flows more intelligently.

Figure 6.8 shows these results. For the first 30 seconds, SFA uses cellular as its measured speed is better than its WiFi constraint (~ 1 Mbps). When the user policy constrains cellular to 640 kbps, however, SFA migrates to WiFi, equally sharing the WiFi bandwidth with MFA, despite the latter’s 5 flows. Once the constraint on cellular is lifted, SFA moves back to cellular. Not only have we achieved fairness when sharing a link, but app policy enables the single app to achieve better performance by responding to the constraints.

Dynamic App Prioritization

In some cases, prioritizing certain app network usage is more desirable for a user than equally sharing resources. Congested or slower cellular links may not have enough bandwidth for simultaneously syncing photographs with the cloud (background) *and* web browsing (foreground). With Tango, a user policy can dynamically prioritize available bandwidth to the foreground app, demonstrated in Figure 6.9. At time 30s, the user opens an app in the foreground. The policy strictly prioritizes the foreground app, giving it the full link rate until the app closes at 60s.

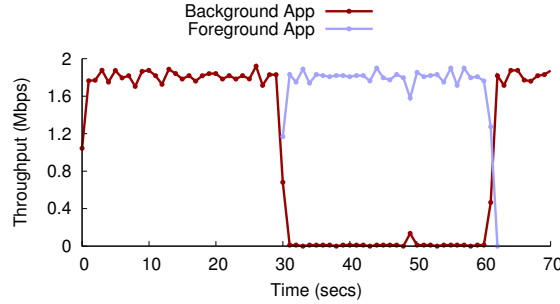


Figure 6.9: Providing priority dynamically to foreground traffic for better user experience.

Fairness, Priority, and App Hints

Tango allows app policies to send hints to the user policy, which provides a powerful way to improve prioritization across apps. The previous example’s policy works on the assumption that the foreground app is always more important than the background app to the user. However, this is not always true, e.g., music streaming in the background. When the music app’s playback buffer gets low, its network usage becomes important to prevent a user from experiencing playback pauses (and thus a poor experience). This is a prime opportunity to use Tango’s app hints.

To demonstrate this, we combine the app-level fairness (i.e., equal bandwidth when both apps are active) and prioritization from the previous examples with app hints. We have two apps competing for a 550 kbps WiFi link: a web app that continually downloads `yahoo.com`’s frontpage (including any embedded or Javascript-initiated content) and our music streaming app. Figure 6.10 shows how this situation performs on today’s smartphones. The page load times, as measured by Android’s WebView, are low until the music app begins in the background. Once that happens, the page load times and variability increase, while the music indiscriminately adds to its playback buffer. The long-running music app is able to fill kernel queues, hindering the burstier web app from getting its fair share, e.g., due to losses and delays in TCP slow start. This lack of resource isolation makes using multiple apps on the phone a poor experience.

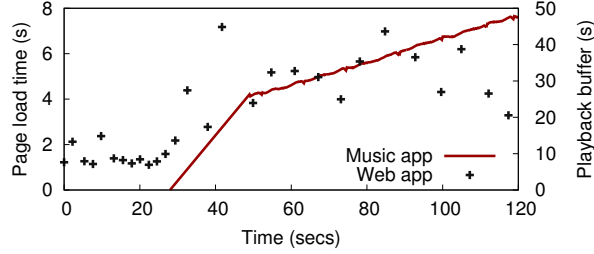


Figure 6.10: Insufficient resource isolation in today’s phones: background music reduces web performance.

Tango can address this problem with its policy enforcement. We implemented a user policy that has two priority classes—high and normal—and equally shares network resources between apps of the same class. Since its in the foreground, the web app always runs at high priority to provide a low-delay experience while browsing. The music app policy sends a hint to the controller that it wants high priority whenever its playback buffer falls below 20s; once the buffer is above 30s, the app sends a hint for normal priority, as its network needs are less urgent. The user policy uses these hints to set the priority constraints for the music app.

Figure 6.11 shows the result of running both apps with this policy. The white area is when the web app runs alone, lightly shaded areas have both apps are running at high priority, and darker shaded areas have the music app is running at normal priority. When the web app runs alone or the music app runs at normal priority, the page load times remain low, as the web app is strictly prioritized over the music app. Page load times only increase when the music app needs to replenish its buffer, hints for a higher priority, and is granted high priority by the user policy. But the load time increase is more modest and less variable than in Figure 6.10, as the equal sharing between the two high-priority apps gives each their own queue, reducing packet loss for the web app. Given the work-conserving nature of the HTB setup, the music app also improves its download rate when the web app stops at time 360s. In short, Tango’s hints allows apps to intelligently cooperate in order to improve overall user experience.

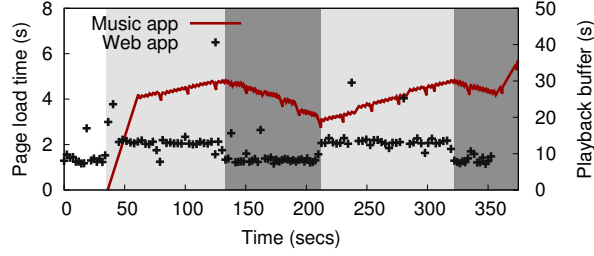


Figure 6.11: Providing dynamic priority between music and web. The music app hints at its need for the network to provide improved page load times when its buffer is healthy (dark gray) and minimal disruption when its buffer needs replenishing (light gray).

These exemplify just a few policy decisions that are useful *across* apps. Tango’s constraint mechanism allows limits to be adjusted on a per-interface basis as well, enabling users to set different management strategies based on the network type. These techniques generalize: user policies can also deal with classes of apps and foreground/background status. Further, apps can optimize in the presence of constraints that restrict their network usage. Non-critical flows can be deferred or given the lowest priority within the app, so that they only consume network resources after other flows complete.

Chapter 7

Conclusion

This thesis provides solutions to updating mobile devices to make network mobility seamless and well-supported, and for improving managing their limited resources.

Through our work on ECCP and Serval, we show that mobility can be achieved with some better separation on network layers and improved naming abstractions. Instead of overloading IP and ports, we introduce serviceIDs for naming applications, flowIDs for naming network flows, and allow IPs to return to their initial purpose of naming physical endpoints. Separating the transport layer into a control and data plane, operating with their own sequence spaces, allows connection control more flexibility to support mobility and multipath. The ECCP protocol has been formally verified [2] to be free of deadlocks and livelocks, and our working Serval implementation has been tested in a variety of mobility use cases to demonstrate its effectiveness and performance (on-par with legacy TCP/IP).

Tango addresses the challenges that come with resource-constrained devices that are exposed to a diversity of network and operating conditions. Tango provides a platform for the many actors (i.e., users and apps) of a device to contribute and shape its resource usage. A global user policy helps device usage in broad strokes and provide the constraints in which app policies can further refine and improve

their usage to align with user expectations. While important for devices today, it is particularly important for devices that make mobility and multipath seamless (e.g., ECCP/Serval-enabled devices), where choice of network can drastically change usage profiles. Our prototype showed improved resource usage in a variety of situations, including saving data for users and improving performance when running multiple apps. Taken altogether, ECCP/Serval and Tango provide a powerful two-pronged approach to improving today’s mobile devices.

7.1 Future Work

While ECCP/Serval and Tango provide strong improvements for dealing with mobility today, several open questions remain.

Multipath support: ECCP supports multipath as part of its protocol, but we did not implement it in our Serval prototype. Adding and removing subflows are possible additional actions to be utilized in Tango, but again, these were not implemented in our first prototype. Multipath would add a new aspect for Tango policy writers to consider. In the context of mobile devices, utilizing multiple networks in the common case may not make sense due to the increased power consumption. But, it could be used effectively in short bursts when network switches are expected. For example, establishing a subflow on a cellular network before shutting down the WiFi could improve handoff times. Both interfaces would only be up for a short time (e.g., when a user policy determines it may need to switch), thereby minimizing the battery hit. There is precedent for this kind of use case, given Apple’s use of MPTCP for its services, e.g., for Siri a user does not have to repeat their request because of a network switch [34].

Other resource management: In Tango we focused on managing network resource usage, mainly looking at things like battery and network data. We would

like to incorporate other resources on the device, such as CPU and memory, into the policy platform as well. Integrating techniques like Linux `cgroups` and CPU Governors [6] into Tango could prove useful in giving users more control over the performance of apps on their devices.

Policy building: While we examined several different approaches to writing Tango policies and targeted several different aspects, the policy space is still quite vast. With new sensors and sources of input to policies being added all the time, the potential policies that could be written also increases. For example, integrating input from devices like wearables (e.g., the Apple Watch, Android Wearables) or dedicated, low-power hardware [33] can help determine what a user is currently doing. This can influence device policy: users who are moving quickly (running, biking, driving) can not easily connect to WiFi points so the phone should be more conservative and have a higher threshold for signal strength.

Bibliography

- [1] Anna Aleryd. How Sony’s Battery STAMINA Mode works. <http://developer.sonymobile.com/2013/04/03/how-sonys-battery-stamina-mode-works/>, April 2016.
- [2] Matvey Arye, Erik Nordström, Robert Kiefer, Jennifer Rexford, and Michael J. Freedman. A Formally-Verified Migration Protocol For Mobile, Multi-Homed Hosts. In *IEEE International Conference on Network Protocols*, October 2012.
- [3] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting Mobile 3G Using WiFi. In *MobiSys*, June 2010.
- [4] Jakob E. Bardram. The Java Context Awareness Framework (JCAF) - A service infrastructure and programming framework for context-aware applications. In *Pervasive Computing*, May 2005.
- [5] D. J. Bernstein. Syn cookies. <http://cr.yp.to/syncookies.html>.
- [6] Dominik Brodowski. Linux CPUFreq: CPUFreq Governors. <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>, April 2016.
- [7] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. In *IEEE Transactions on Software Engineering*, October 2003.
- [8] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in Cyberspace: Defining Tomorrow’s Internet. *IEEE/ACM Trans. Netw.*, 13(3), June 2005.
- [9] Patrick Fahy and Siobhan Clarke. CASS - A middleware for mobile context-aware applications. In *Workshop on Context Awareness at MobiSys*, June 2004.
- [10] Dino Farinacci, Vince Fuller, Dave Meyer, and Darrel Lewis. RFC 6830: The Locator/ID Separation Protocol (LISP), January 2013.
- [11] Bryan Ford and Janardhan Iyengar. Breaking up the transport logjam. In *HotNets*, October 2008.
- [12] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. In *IEEE International Conference on Network Protocols*, October 1997.

- [13] Brett Higgins, Jason Flinn, T.J. Giuli, Brian Noble, Christopher Peplin, and David Watson. Informed Mobile Prefetching. In *MobiSys*, June 2012.
- [14] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, May 1997.
- [15] Amazon Inc. Amazon Web Services (AWS). <https://aws.amazon.com/>, December 2015.
- [16] Apple Inc. Background Execution. <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html>, April 2016.
- [17] Google Inc. Optimizing for Doze and App Standby. <http://developer.android.com/training/monitoring-device-state/doze-standby.html>, April 2016.
- [18] Microsoft Inc. Background agents for Windows Phone 8. [https://msdn.microsoft.com/en-us/library/windows/apps/hh202942\(v=vs.105\).aspx](https://msdn.microsoft.com/en-us/library/windows/apps/hh202942(v=vs.105).aspx), April 2016.
- [19] Robert Kiefer, Erik Nordström, and Michael J. Freedman. From feast to famine: Managing mobile network resources across environments and preferences. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, October 2014.
- [20] Leslie Lamport. Password Authentication with Insecure Communication. In *Communications of ACM*, November 1981.
- [21] Kyunghan Lee, Injong Rhee, Joohyun Lee, Song Chong, and Yung Yi. Mobile Data Offloading: How Much Can WiFi Deliver? In *CoNEXT*, November 2010.
- [22] Anthony J. Nicholson and Brian D. Noble. BreadCrumbs: Forecasting mobile connectivity. In *MOBICOM*, September 2008.
- [23] Pekka Nikander, Andrei Gurtov, and Thomas R. Henderson. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *IEEE Comm. Surveys*, 12(2), April 2010.
- [24] Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer, Matvey Arye, Steven Ko, Jennifer Rexford, and Michael J. Freedman. Serval: An End-Host Stack for Service-Centric Networking. In *NSDI*, April 2012.
- [25] Olga Ormond, John Murphy, and Gabriel-Miro Muntean. Utility-based Intelligent Network Selection in Beyond 3G Systems. In *IEEE ICC*, June 2006.
- [26] Christophe Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *Cell-Net*, August 2012.

- [27] Charles E. Perkins. RFC 3344: IP mobility support for IPv4, August 2002.
- [28] Charles E. Perkins and David B. Johnson. Mobility support in IPv6. In *MOBICOM*, November 1996.
- [29] Brandon Podmayersky. An incremental deployment strategy for Serval. Technical Report TR-903-11, Princeton CS, June 2011.
- [30] Moo-Ryong Ra, Jeongyeup Paek, Abhishek B. Sharma, Ramesh Govindan, Martin H. Krieger, and Michael J. Neely. Energy-Delay Tradeoffs in Smartphone Applications. In *MobiSys*, June 2010.
- [31] Zhijie Shi, Chujiao Ma, Jordan Cote, and Bing Wang. Hardware Implementation of Hash Functions. In Mohammad Tehranipoor and Cliff Wang, editors, *Introduction to Hardware Security and Trust*. Springer-Verlag New York, 2012.
- [32] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, August 2000.
- [33] Andrew Tarantola. Google’s Android Sensor Hub knows how your Nexus is moving. <http://www.engadget.com/2015/09/29/google-android-sensor-hub/>, April 2016.
- [34] Iljitsch van Beijnum. Multipath TCP lets Siri seamlessly switch between Wi-Fi and 3G/LTE. <http://arstechnica.com/apple/2013/09/multipath-tcp-lets-siri-seamlessly-switch-between-wi-fi-and-3glte/>, April 2016.
- [35] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *NSDI*, March 2004.
- [36] <http://web10g.org/>, April 2016.
- [37] Ashton L. Wilson, Andrew Lenaghan, and Ron Malyan. Optimising Wireless Access Network Selection to Maintain QoS in Heterogeneous Wireless Environments. In *WPMC*, September 2005.
- [38] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *NSDI*, March 2011.
- [39] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Z. Morley Mao, Jeffrey Pang, and Shobha Venkataraman. Identifying Diverse Usage Behaviors of Smartphone Apps. In *IMC*, November 2011.
- [40] Kok-Kiong Yap, Te-Yuan Huang, Masayoshi Kobayashi, Yiannis Yiakoumis, Nick McKeown, Sachin Katti, and Guru Parulkar. Making Use of All the Networks Around Us: A Case Study on Android. In *CellNet*, August 2012.

- [41] Jukka Ylitalo, Tony Jokikyyny, Tero Kauppinen, Antti J. Tuominen, and Jaakko Laine. Dynamic network interface selection in multihomed mobile hosts. In *HICSS*, January 2003.
- [42] Shelley Zhuang, Kevin Lai, Ion Stoica, Randy Katz, and Scott Shenker. Host mobility using an internet indirection infrastructure. In *MobiSys*, May 2003.